NASA Contractor Report 181732

Application Developer's Tutorial for the CSM Testbed Architecture

Phillip Underwood and Carlos A. Felippa

Lockheed Missiles and Space Company, Inc.
Palo Alto, California

Contract NAS1-18444

October 1988

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

1. Introduction

This tutorial is an extension of [1]† Appendix D to include a simple interface to GAL-DBM (Global Access Library - Database Management), the database management system for the CSM Testbed Architecture. GAL-DBM is described in [2]. The goal is to present a complete, but simple, introduction to using both CLIP (Command Language Interface Program) and GAL to write a NICE Processor. To achieve this goal the first author has extended the second author's work to include the interface to GAL. Much of the previous text describing commands and CLIP has been retained to make the tutorial stand alone.

Before beginning this tutorial, you should be familiar with the CSM Testbed Architecture (NICE). As a minimum you should: read [1] Appendices C & D — make sure you understand the use of the CLIP entry points, which may require a reading of [1]; and read [2] — at least be familiar with the ideas for nominal datasets.

The example Processor presented here is still quite simple as production Processors go, but is no longer trivial. It requires one to two weeks to put together. The Processor solves a two-dimensional elastostatic problem by a directly-formulated* Boundary Element Method (BEM), and is named, appropriately, DBEM2.

The "kernel" of the Processor is a BEM-program adapted from the book *Boundary Elcment Methods in Solid Mechanics* by S. L. Crouch and A. M. Starfield [4]. The program is called TWOBI and is presented in Appendix C of the book; it is based on the boundary-integral theory covered in Section 6 therein.

The program is appropriate as an example of the use of interactive techniques because the input data are fairly simple but the commands are of multiple-item type and thus serve to illustrate things like phrases, item lists, qualifiers, and defaults. The program is somewhat weak for illustrating the true use of a database. However, it is relatively simple, so we can concentrate on the mechanics of using GAL. Ways to extend DBEM2 and use GAL for more complex problem solving are discussed as we proceed through this example Processor.

To prepare the reader for subsequent sections we need to cover some background material on the GAL-DBM [2].

The two-level conceptual model of GAL must be understood; see [2], §2.2. The two-levels of data in nominal GAL are named datasets and named records. The GAL database file is usually called a Library. In this Library are books (datasets) and in the books are chapters (records). In this tutorial example we will only consider one active Library; however, complex Processors can have several active GAL-DBM Libraries. We will have several datasets and several associated records in our one Library.

† Numbers in brackets refer to references at end of report.

* The term *direct formulation* refers to the technique used in deriving the governing boundary-integral equations. Direct methods are formulated from the start in terms of physical quantities such as displacement and stress fluxes. On the other hand, indirect methods are formulated in terms of source strength distributions, which have no direct physical meaning and are eventually eliminated following spatial discretization.

The two-level conceptual model of GAL must be understood; see [2], §2.2. The two-levels of data in nominal GAL are named datasets and named records. The GAL database file is usually called a Library. In this Library are books (datasets) and in the books are chapters (records). In this tutorial example we will only consider one active Library; however, complex Processors can have several active GAL-DBM Libraries. We will have several datasets and several associated records in our one Library.

Datasets are usually chosen as functional groups of data records. If you are familiar with the programming language C, a dataset is analogous to a structure and GAL records are just like member definitions within a structure. In the boundary element method and in other discrete element methods, such as finite element methods, typical functional groups of data are geometry, material, boundary conditions, loadings, elements, system matrices (coefficient or stiffness), system vectors (right-hand-side(s), solution(s), etc.), stress/strain/resultants, etc.

For example, we may decide to have a dataset named `GEOMETRY` and in this dataset there may be records named `NUMBER_NODES`, `NODES`, `COORDS` (or `X-COORDS`, `Y-COORDS`, ...), etc. For this tutorial the geometry data is stored in a dataset named `SEGMENT`; see §3.1.

In this tutorial, we will use fixed or "hardwired" names for the datasets and records. This simplifies the Processor and lets us get on with how to use GAL, without getting into the complex issues of tables and their management to relate Library dataset names and record names to the names used internally in the Processor. In addition, the command set would have to be expanded to include commands to bind the external database names with the internal Processor names. In general fixed names work well with tightly coupled Processors, because they don't interact much with other Processors. The fixed names also make the Processor much easier for the user to run. The user doesn't have to remember as many commands or keep track of where the data really are. Loosely coupled networks of Processors may need the capability to use datasets and records of any given name. However, even fixed dataset and record names can be changed by using the **`*rename dataset`** and/or the **`*rename record`** directives; see [3], §53.1 & §53.2.

Another GAL-DBM feature is word addressability. With this feature a particular entry in a dataset-record can be extracted or stored [2], §5.1. For example, the *i-th* entry for the geometry nodes, `X-COORDS & Y-COORDS`, can easily be extracted. This feature can be used for out-of-core techniques. In actual practice experienced Processor developers use a local dynamic memory manager for out-of-core methods, because they are usually more efficient than GAL. GAL was designed for efficient use of archival data. These are advanced concepts. They are not covered in this tutorial, but the developer should be aware of these issues — especially for large problems.

3. The Data Structures

Following sound practice, we begin by designing the data structures. The task is more complicated for DBEM2 than for the simple program in [1] Appendix C. We will retain DBEM2 as a single Processor, but add a global database that corresponds to the data structures and functions in DBEM2. Possible avenues to explore in separating DBEM2 into several Processors are presented, but not pursued in depth. The main use of the database in this example is to archive problem data for a restart, archive problem data to document what was done, and archive problem data so that some old data can be used with new input to solve a slightly different problem. Our main goal, to illustrate the mechanics of using GAL, is well served by this approach.

The task is simplified by the following considerations:

1. The Processor presented here is isolated from others. There is no need to transact business with a global database generated by other Processors.
2. DBEM2 makes use of only one matrix, which is generally unsymmetric and full. There being no need to make use of sparse storage formats, an ordinary FORTRAN array suffices.
3. Everything is assumed to fit in core at one time. Not having to deal with auxiliary storage avoids many complications.
4. The internal data structures and the GAL dataset-record structures are the same.

All data that have to be shared among many parts of DBEM2 are accommodated in labelled common blocks. The first author is not in favor of using labelled common blocks for moving data from one subroutine to another, but it is retained here for expediency. Thus in the present Processor several blocks are used to group data according to function. Furthermore, the blocks are declared in separate files whose extension is `inc`. These files are inserted where they are needed via `INCLUDE` statements. The use of `INCLUDE` enforces consistency (everything is declared only once) and makes maintenance and modification much easier.

Remember that in §1.0 we said we would use fixed dataset names and record names. Thus, here we will use, wherever possible, the labelled common block name for the dataset name and the variable names in the labelled common block will be the same as the record names. In a few cases we will have to break one labelled common block into two or three separate datasets to achieve a realistic functional group for the data. In these cases we create new, but hopefully meaningful, names.

3.1 The Segment Data

We begin by setting up the data for boundary segments, which is placed in file **segment.inc**. The maximum number of segments is parameterized to be MAXSEG, which is set to 20 in the version listed below.

The DBEM2 user will be allowed to define segments in any order and give them arbitrary numbers from 1 through MAXSEG, so we need a "marker" array that tells which segments have been defined. We also need a counter of how many boundary elements are in each defined segment. Then there are the geometric arrays: the x and y coordinates of the end points. Finally, there are the boundary condition arrays: one integer code (related to that used by reference 4) and two floating-point arrays of prescribed shear and normal values. Here is a list of the file that groups this information:

```
*
*      This is the file segment.inc
*
       common /SEGMENT DATA/
     $    segdef, numel, xbeg, ybeg, xend, yend, kode, bvs, bvn
       integer    MAXSEG
       parameter (MAXSEG=20)    ! Maximum no. of boundary segments
       integer segdef(MAXSEG)   ! Segment definition tag
       integer numel(MAXSEG)    ! Number of BE divisions of segment
       real    xbeg(MAXSEG)     ! X-coord of starting segment point
       real    ybeg(MAXSEG)     ! Y-coord of starting segment point
       real    xend(MAXSEG)     ! X-coord of ending segment point
       real    yend(MAXSEG)     ! Y-coord of ending segment point
       integer kode(MAXSEG)     ! Segment BC code
       real    bvs(MAXSEG)      ! Prescribed shear value
       real    bvn(MAXSEG)      ! Prescribed normal value
```

The style used in this INCLUDE file will be followed for all others. There is a COMMON declaration that lists the shared variables. Then each variable is declared on a separate line. The variable name is followed by an inline comment that provides a short description of the function of each variable. This brief documentation should be entered at the time you prepare or update the INCLUDE file; if you leave it for later, it'll never be done.

When you get farther into this tutorial you will see that the segment data described here is generated in at least two subroutines and possibly three, if you define non-default data for the number of BE divisions of a segment. Thus to maintain a functional breakdown of data in the database, these data are divided into two datasets: 1) SEGMENT with records named SEGDEF, NUMEL, XBEG, YBEG, XEND, and YEND that hold **segdef**, **numel**, **xbeg**, **ybeg**, xend, and yend one-dimensional array data; and 2) BCVALUES with records named KODE, BVS, and BVN that hold the kode, bvs, and bvn one-dimensional data. The default value for **numel** is 1, however you may enter other values by using the **DEFINE ELEMENTS** command. Data generated by this command are written over the default data created in the DEFINE SEGMENTS code.

Although this simple structure for the datasets and records may seem trivial, it is quite common even in a complex Processor. That is, the data in the database are often structured just as the data are used in the code. Also, note that, the two level structure of GAL, named datasets with named records, lends itself to a functional grouping of the data with names that are easy to relate to the data generated and used by the Processor. This simple structure for the datasets and the records will be used throughout the DBEM2 Processor.

In summary, we have defined two datasets with their associated records as:

```
Dataset - SEGMENT
          Records - SEGDEF
                    NUMEL
                    XBEG
                    YBEG
                    XEND
                    YEND
```

and

```
Dataset - BCVALUES
          Records - KODE
                    BVS
                    BVN
```

3.2 The Material Data

Since we are dealing with a homogeneous elastic isotropic material and we ignore thermal effects, the material is fully characterized by two properties: the elastic modulus E and the Poisson's ratio ν. These two are collected in file `material.inc` :

```
*
*     This is the file material.inc
*
      common /MATERIAL/  em, pr
      real     em     ! Elastic modulus
      real     pr     ! Poisson's ratio
```

Here the entry for the database is very simple to design. We use a dataset named MATERIAL with two records named EM and PR to store the values for em and pr. Thus, this dataset has the following two level structure:

```
Dataset - MATERIAL
          Records - EM
                    PR
```

3.3 The Symmetry Data

The program allows one or two lines parallel to the coordinate axes to be specified as axes of symmetry. For example, $x = 2.5$ or $y = -1.50$, or both. Three pieces of data accommodate this information: one symmetry tag (0=none, 1=symmetry about $x = a$, 2 = symmetry about $y = b$, 3 = double symmetry), and the values of a and b as appropriate. The necessary declarations are placed in the file `symmetry.inc` :

```
*
*     This is the file symmetry.inc
*
      common /SYMMETRY_DATA/
     $    ksym, xsym, ysym
      integer   ksym         ! symmetry tag
      real      xsym, ysym   ! symmetry about x=a & y=b values
```

These data are very similar to the `MATERIAL` dataset above, so we choose a similar design for the `SYMMETRY` dataset and records. Here it is:

```
Dataset - SYMMETRY
          Records - KSYM
                    XSYM
                    YSYM
```

3.4 The Prestress Data

The program allows a constant initial-stress field to exist in the *undeformed* medium. This *prestress* tensor field is defined by the three components σ^0_{xx}, σ^0_{yy} and σ^0_{xy}. If undefined, these three values are assumed to be zero. File **prestress.inc** contains the appropriate declarations:

```
*
*       This is the file prestress.inc
*
        common /PRESTRESS/  sxx0, syy0, sxy0
        real     sxx0    ! Prestress (initial field stress) sigma_xx
        real     syy0    ! Ibid., for sigma_yy
        real     sxy0    ! Ibid., for sigma_xy
```

Prestress data are especially important for analysis of *unbounded domains*, for which they assume the role of conditions at infinity. For example, suppose that we want to analyze the effect of a hole in an infinite region under uniform uniaxial stress, say $\bar{\sigma}_{xx}$. Then we set $\sigma^0_{xx} = \bar{\sigma}_{xx}$, $\sigma^0_{yy} = \sigma^0_{xy} = 0$ in the input data.

Again very similar to MATERIAL and SYMMETRY, so we have:

```
Dataset - PRESTRESS
          Records - SXX0
                    SYY0
                    SXY0
```

3.5 The Element Data

The most voluminous data are that pertaining to the boundary elements, since typically there will be many elements per segment. The information is collected in file **element.inc** , which reads

```
*
*     This is the file element.inc
*
      common /ELEMENT DATA/
     $   numbe, xme, yme, hleng, sinbet, cosbet, kod, c, b, r, x
      integer    MAXELM, MAXEQS
      parameter (MAXELM=100)     ! Maximum no. of boundary elements
      parameter (MAXEQS=2*MAXELM) ! Maximum no. of discrete equations
      integer    numbe           ! Total number of boundary elements
      real       xme(MAXELM)     ! X-coor of element midpoint
      real       yme(MAXELM)     ! Y-coor of element midpoint
      real       hleng(MAXELM)   ! Half length of element
      real       sinbet(MAXELM)  ! Sine of (element,x) angle
      real       cosbet(MAXELM)  ! Cosine ibid.
      integer    kod(MAXELM)     ! Elem BC code (copies seg code)
      real       b(MAXEQS)       ! Prescribed boundary values
      real       c(MAXEQS,MAXEQS) ! Influence coefficient matrix
      real       r(MAXEQS)       ! Forcing (RHS) vector
      real       x(MAXEQS)       ! Solution vector
```

The elements arrays such as xme, yme, etc are parametrized in terms of the maximum number of elements MAXELM.

This block also contains arrays used to set up and solve the BEM equation system, namely c, r, b and x. These are parameterized in terms of the total number of equations MAXEQS, which of course is twice MAXELM.

Now things get a little more complicated. All these data could be stored in one dataset with several records, but it is better to use a more functional design as in §3.1. So, the data in the database are organized according to where it is generated. All the actual element data are computed in the BUILD subroutine, so the first dataset of this group is named ELEMENT with records named NUMBE, XME, YME, HLENG, SINBET, COSBET, KOD, and B that hold the integer number **numbe**, and the one-dimensional arrays **xme, yme, hleng, sinbet, cosbet,** kod, and b. These data are analogous to the element stiffness data in a typical finite element code. However, for boundary elements there are no individual element stiffnesses, only a global system coefficient matrix. The element data just contain the information needed to compute the global coefficient matrix.

The data for the arrays c and r are created in the GENERATE subroutine. However, in general the system matrix, analogous to a global stiffness matrix in a finite element code, is computed in one subroutine, such as an assembler, and the right-hand-side, the forcing function, is computed in another subroutine. Thus, two datasets are added to the database for these data. The first dataset is named COEFF with a record named C that contains the data for the coefficient matrix, c. The second dataset is named RHS with a record named R that contains the one-dimensional array r.

The final dataset for this group of data is for the solution vector, x. This one-dimensional array is computed by SOLVE, very similarly to a typical finite element code. So, we name the dataset SOLUTION with a named record X to archive the solution vector x.

In summary, we have defined four datasets with their associated records as:

```
Dataset - ELEMENT
            Records - NUMBE
                      XME
                      YME
                      HLENG
                      SINBET
                      COSBET
                      KOD
                      B
Dataset - COEFF
            Record  - C
Dataset - RHS
            Record  - R
  and
Dataset - SOLUTION
            Record  - X
```

3.6 The Field Location Data

This block of data pertains to the location of field points at which stresses and displacements are to be calculated once the boundary solution is obtained. The program allows these locations to be specified as equally spaced points along straight lines defined by the user. Up to MAXLIN (= 100 in the version below) lines can be defined. The locations are specified by giving the x and y coordinates of the first and last points on the line, and the number of intermediate points (≥ 0) to be "collocated" between the first and last points. An isolated point may be specified by making the first and last point coincide.

All of this information is gathered in the file **output.inc** :

```
*
*     This is the file output.inc
*
      common /OUTPUT DATA/
     $   lindef, nintop, xfirst, yfirst, xlast, ylast
      integer    MAXLIN
      parameter (MAXLIN=100)
      integer   lindef(MAXLIN)   ! Line definition tag
      integer   nintop(MAXLIN)   ! No. of intermediate points on line
      real      xfirst(MAXLIN)   ! X-coor of first point on line
      real      yfirst(MAXLIN)   ! Y-coor of first point on line
      real      xlast(MAXLIN)    ! X-coor of last point on line
      real      ylast(MAXLIN)    ! Y-coor of last point on line
```

This group of data are similar to other groups of one-dimensional arrays, such as the segment data in §3.1. So, we use a dataset named FIELD with named records LINDEF, NINTOP, XFIRST, YFIRST, XLAST, and YLAST to store the one-dimensional arrays **lindef**, **nintop**, **xfirst**, **yfirst**, **xlast**, and **ylast**. In outline form this database data structure is:

```
Dataset - FIELD
          Records - LINDEF
                    NINTOP
                    XFIRST
                    YFIRST
                    XLAST
                    YLAST
```

3.7 The Database Data

The DBEM2 Processor subroutines that communicate with the GAL-DBM need to know the logical device index (ldi) of the library (database) being used; see [2] §2.4 and the description of the DB_OPEN subroutine in §6. So, this globally used information is kept in the labelled common block DATABASE. There is no dataset and record associated with these data because it is not archival data; it is temporary — only used for the run at hand.

This information is gathered in the file **database.inc** :

```
*
*     This is the file database.inc
*
      common /DATABASE/  ldi
      integer    ldi    ! GAL Library logical device index
```

This concludes the design of the important data structures for the internal data representation and the global database. Next we pass to the design of a command set to control logic of DBEM2.

4. The Commands

Having described the data and the datasets for the database, we have now to design an appropriate set of commands to perform operations on the data and the database. The writers found it convenient to choose commands headed by the following action verbs:

```
CLEAR
OPEN
DEFINE
BUILD
GENERATE
SOLVE
PRINT
CLOSE
STOP
```

Why these particular commands? Partly from a preliminary study of the problem, partly from wishes to get several command formats so that the use of many of the entry points described in [1] would be illustrated.

It turns out that the last wish (of illustrating various command formats) makes the command set a bit inconsistent, but that should not cause a great deal of concern. After all, it's only an example.

Another Processor developer faced with the same problem (even a simple problem like this one) may in fact come up with a radically different set of commands that accomplishes virtually the same thing.

We next describe briefly what the commands do.

`CLEAR`	Initializes all Tables maintained by the Processor and sets some default values.
`OPEN`	Opens a GAL-DBM Library to store problem data and/or load previously stored problem data.
`DEFINE`	Enters data that are used in the definition of the problem to be solved. The `DEFINE` verb will be followed by another keyword that makes the data more specific.
`BUILD`	Indicates that the problem-definition phase is complete, and calls for the generation of the discrete governing equations.
`GENERATE`	Triggers the assembly of the influence coefficient matrix and forcing vector.
`SOLVE`	Triggers the solution for the unknown boundary variables.
`PRINT`	Prints displacements and stresses at boundary points and at specified field points.
`CLOSE`	Closes the open GAL-DBM Library.
`STOP`	Closes all open GAL-DBM Libraries and terminates execution of the processor.

THIS PAGE LEFT BLANK INTENTIONALLY.

5. Starting at the Top

We are going to build the Processor Executive "top down". For this relatively small Processor it probably doesn't make much difference whether we do it top-down, bottom-up or inside-out. But adhering to this approach makes life easier for bigger Processors.

Following the top-down approach we must do the main program first. Here it is:

```
*
*     Computer Program for the Two-Dimensional Direct
*     Boundary Element Method (DBEM2)
*
*     Adapted from program TWOBI in the book Boundary Element Methods
*     Methods in Solid Mechanics by S. L. Crouch and A. M. Starfield,
*     G. Allen & Unwin, London, 1983,  by Philip Underwood and
*     C. A. Felippa to exemplify conversion to interactive operation
*     via CLIP and the use of a global database GAL-DBM.
*
      program    DBEM2
*
      implicit      none
      character*8   CCLVAL, verb
      integer       ICLTYP
*
 1000   call      CLREAD (' DBEM2> ',
     $          '  CLEAR, OPEN, DEFINE, BUILD[/LOAD | STORE]&& '//
     $          'GENERATE[/LOAD | STORE], SOLVE[/LOAD | STORE]&& '//
     $          'PRINT, CLOSE, STOP')
        if (ICLTYP(1) .le. 0)   then
          print *, '*** Commands must begin with keyword'
        else
          verb =   CCLVAL(1)
          call     DO_COMMAND (verb)
        end if
        go to 1000
      end
```

In DBEM2 the top-level command must start with an action verb, hence the error check. The prompt is the name of the Processor: this is a convention followed in the NICE system.

The top level of all Processors looks very much the same, regardless of the complexity of what lies underneath. This is not surprising if you note that all Processors fit the "do forever" model illustrated in [1] §C.2.

The next level is DO_COMMAND, which is again a "case" statement that branches on the action verb:

```
*
*     Top level command interpreter for DBEM2
*
      subroutine    DO_COMMAND  (verb)
*
      implicit      none
      character     key*8, qual*8, verb*(*)
      integer       nq
      logical       CMATCH
*
      key =   verb
      if (CMATCH (key, 'B^UILD'))          then
        qual = ' '
        call   CLOADQ (' ', -1, qual, 0, nq)
        if ( nq .eq. 1 ) then
          call UPCASE (qual)
          if ( qual(1:1) .eq. 'L' ) then
            qual = 'LOAD'
          else if ( qual(1:1) .eq. 'S' ) then
            qual = 'STORE'
          else
            print*, ' Illegal qualifier: ', qual, ' for BUILD.'
            print*, ' BUILD not performed.'
            return
          end if
        end if
        call  BUILD ( qual )
      else if (CMATCH (key, 'CLE^AR'))     then
        call  CLEAR
      else if (CMATCH (key, 'CLO^SE'))     then
        call  DB_CLOSE
      else if (CMATCH (key, 'D^EFINE'))    then
        call  DEFINE
      else if (CMATCH (key, 'G^ENERATE')) then
        qual = ' '
        call   CLOADQ (' ', -1, qual, 0, nq)
        if ( nq .eq. 1 ) then
          call UPCASE (qual)
          if ( qual(1:1) .eq. 'L' ) then
            qual = 'LOAD'
          else if ( qual(1:1) .eq. 'S' )  then
            qual = 'STORE'
          else
            print*, ' Illegal qualifier: ', qual, ' for GENERATE.'
            print*, ' GENERATE not performed.'
            return
          end if
        end if
        call  GENERATE ( qual )
      else if (CMATCH (key, 'H^ELP'))     then
        call  HELP
      else if (CMATCH (key, 'O^PEN'))     then
        call  DB_OPEN
      else if (CMATCH (key, 'P^RINT'))    then
        call  PRINT
      else if (CMATCH (key, 'SO^LVE'))    then
        qual = ' '
        call   CLOADQ (' ', -1, qual, 0, nq)
        if ( nq .eq. 1 ) then
          call UPCASE (qual)
          if ( qual(1:1) .eq. 'L' ) then
            qual = 'LOAD'
          else if ( qual(1:1) .eq. 'S' )  then
            qual = 'STORE'
          else
            print*, ' Illegal qualifier: ', qual, ' for SOLVE.'
            print*, ' SOLVE not performed.'
            return
          end if
        end if
        call  SOLVE ( qual )
      else if (CMATCH (key, 'ST^OP'))     then
        call  STOP
      else
        print *, '*** Illegal or ambiguous verb: ', key
      end if
      return
      end
```

Note that the tests are ordered so that keywords are alphabetically sorted. This makes it easier to insert new keywords without forgetting to expand roots of existing ones. For example, suppose you want to insert a PLOT command for your favorite graphic device; inserting it just before the test for PRINT makes it easy to spot that the root for the latter has to be expanded to PR.

For the keywords, BUILD, GENERATE, and SOLVE we are also looking for a possible qualifier. The existence of this qualifier is determined by a call to CLOADQ; see [1] §8.3. If a qualifier is found, then we check to see if the value is either LOAD or STORE. The LOAD qualifier signifies that the data are to be loaded from the database instead of being computed. The STORE qualifier signifies that the data are to be stored in the database after it has been computed.

UPCASE is a CSM Testbed (NICE) architectural utility that converts its argument to uppercase.

THIS PAGE LEFT BLANK INTENTIONALLY.

6. Starting and Stopping

The **CLEAR** subroutine is quite simple, as it only has to zero out the model definition tables:

```
*
*     Initialize tables, set default values
*
      subroutine    CLEAR
C
      implicit      none
      include       'segment.inc'
      include       'element.inc'
      include       'material.inc'
      include       'symmetry.inc'
      include       'prestress.inc'
      include       'output.inc'
      integer       i
*
      do 1500  i = 1,MAXSEG
        segdef(i) = 0
        xbeg(i) =  0.0
        xend(i) =  0.0
        ybeg(i) =  0.0
        yend(i) =  0.0
        numel(i) =  0
        kode(i) =   0
        bvs(i) =   0.0
        bvn(i) =   0.0
 1500   continue
*
      do 2000  i = 1,MAXLIN
        lindef(i) = 0
 2000   continue
      numbe =  0
*
      ksym =   0
      em =     1.0
      pr =     0.0
      sxx0 =   0.0
      syy0 =   0.0
      sxy0 =   0.0
      print *, 'Tables initialized'
      return
      end
```

The function of the arrays is explained in §3.

Next, you may wish to OPEN a GAL Library (database) to use for loading previously computed data and/or to store data computed during a run. (CLEAR does not have to be used before OPEN and vice versa.) The OPEN command has the form

```
OPEN/[Qualifier]    LIB = library_name
```

The brackets, [], around the Qualifier signify that the Qualifier is optional. The Qualifier is to describe the characteristics of the Library to be opened. The accepted values are NEW, OLD, ROLD, and SCR. If no qualifier is given the value defaults to COLD. The meaning of these values is given in [2], Table 6.5. Most applications may ignore the qualifier.

The name of the file that is the GAL Library (library_name) must be entered. This is a valid file name for the computer system you are using. On a UNIX system if you use a pathname that contains /'s the file name must be enclosed in single quotes (e.g., '/usr/king/kong/new/york'). Otherwise CLIP will try to interpret the directories and files as qualifiers.

Experienced NICE users pick file names for GAL Libraries that are descriptive of the problem and usually use the file extension gal. This way you can easily find the database files and the name should remind you of the problem.

The OPEN command produces a call to the DB_OPEN subroutine:

```
*
*     Open GAL Library (Database)
*
      subroutine     DB_OPEN
*
      implicit       none
      include       'database.inc'
      character*80   CCLVAL
      character*81   libnam
      character*11   key
      character*4    qual
      integer        ICLNIT, ICLSEK, ICLTYP, LENETB, LMOPEN
      integer        nq
*
      if ( ICLNIT ( ) .lt. 3 ) then
        call   CLREAD (' OPEN: Enter [/QUAL] LIB = LIB_NAME > ',
     $                 ' ' )
      end if
*
      libnam = ' '
      key = 'COLD/GAL82 '
      call   CLOADQ (' ', -1, qual, 0, nq)
      if ( nq .eq. 1 ) then
        call UPCASE (qual)
        if ( qual(1:1) .eq. 'N') key = 'NEW/GAL82 '
        if ( qual(1:1) .eq. 'O') key = 'OLD/GAL82 '
        if ( qual(1:1) .eq. 'R') key = 'ROLD/GAL82 '
        if ( qual(1:1) .eq. 'S') key = 'SCR/GAL82 '
      end if
      if ( ICLSEK (0, 'L^IB') .ne. 0 ) then
        if ( ICLTYP (0) .gt. 0 ) libnam = CCLVAL(0)//' '
        if ( libnam .eq. ' ' ) then
          print*,
     $    ' Cannot find LIB_NAME.  Format is LIB = LIB_NAME.'
          print*, ' No library opened.'
          return
        end if
      else
        print*, ' Cannot find keyword LIB; No library opened.'
        return
      end if

      ldi =   LMOPEN ( key, 0, libnam, 0, 500 )
      if ( ldi .eq. 0 ) then
        print*, ' Unable to open library: ', libnam(1:LENETB(libnam))
      else
        call  GMSIGN ( 'DBEM2' )
      end if

      return
      end
```

The call to CLREAD, [1], §2.7, at the beginning of DB_OPEN prompts the user for needed data if the minimum number of items (i.e., the function ICLNIT, [1], §9.5) needed to carry out the OPEN command is not found.

In DB_OPEN the call LMOPEN is used to open the GAL Library for reading (loading) and/or writing (storing) data. The subroutine GMOPEN can also be used, but for this beginning example it is a little complex. The use of LMOPEN here easily lets us use one database for one problem. If multiple databases are required then GMOPEN is the one to use. A description of GMOPEN and LMOPEN can be found in [2], §6.4.

If the Library is successfully opened (a non-zero ldi is obtained) we call GMSIGN with the name of the Processor as the argument. This places the Processor name in the Table of Contents (TOC) data for the database. This is a recommended procedure; see [2], §10.8.

The DB_CLOSE subroutine carries out the action requested by the CLOSE command. This command is used to close the currently active GAL Library. This command is used if you were finished with some problem then wished to work on another problem that required another GAL Library. So you CLOSE the old one before OPENing the new file containing the other GAL Library.

The DB_CLOSE subroutine is quite simple:

```
*
*      Close GAL Libraries (Databases)
*
       subroutine       DB_CLOSE
*
       implicit         none
       include          'database.inc'
*
       call GMCLOS ( ldi, 0, 100 )
*
       return
       end
```

The only call is to GMCLOS [2], §6.2, which closes the active library associated with ldi. For more advanced applications with multiple libraries in use, a specific ldi may also be an input item. So a specific library could be closed after it is no longer needed.

Relatively simple is the STOP subroutine:

```
*
*      Terminate the run
*
       subroutine  STOP
       call   GMCLOS ( 0, 0, 100 )
       print*, ' Hope you enjoyed the ride!'
       call   CLPUT  ( '*stop ' )
       end
```

The call to GMCLOS insures that any GAL Libraries that may be open are properly closed, so that no database data are lost. The call to CLPUT sends the *stop directive to CLIP. This is the preferred way to exit a NICE Processor, because it allows you to run the Processor in network mode. However, if the network mode (SuperCLIP) is not available on your computer, replace this line with the FORTRAN stop statement.

A description of GMCLOS can be found in [2], §6.2. A description of CLPUT can be found in [1], §2.4. And a description of *stop can be found in [3], §59.1.

Now on to DEFINE — to define the problem to solve.

7. Defining the Problem

The DEFINE command introduces problem-definition data. It is convenient to break up the definition into several types of data, which correspond closely to the data-structure grouping discussed in §3. Each type is identified by a keyword that immediately follows DEFINE. The keywords are:

SEGMENTS	Specifies the straight-line segments that make up the boundary of the problem to be solved.
ELEMENTS	Specifies into how many boundary elements each segment will be divided.
BOUNDARY CONDITIONS	Specifies the boundary conditions that apply to each boundary segment.
SYMMETRY CONDITIONS	Specifies the symmetry conditions, if any, that apply to the problem to be solved.
MATERIAL	Specifies constitutive properties of the material.
PRESTRESS	Specifies prestress data in the form of initial stress components.
FIELD	Specifies the location of field points at which displacement and stresses are to be evaluated and printed later.

Subroutine DEFINE, unlike CLEAR, OPEN, CLOSE or STOP, branches as per the second keyword:

```
*
*     Interpret DEFINE command
*
      subroutine    DEFINE
*
      implicit      none
      character     key*8, CCLVAL*8
      integer       ICLTYP
      logical       CMATCH
*
      if (ICLTYP(2) .le. 0)  then
        print *, '*** No keyword after DEFINE'
        return
      end if
*
      key =    CCLVAL(2)
      if (CMATCH (key, 'B^OUND'))          then
        call  DEFINE_BOUNDARY_CONDITIONS
      else if (CMATCH (key, 'E^LEMENTS')) then
        call  DEFINE_ELEMENTS
      else if (CMATCH (key, 'F^IELD'))     then
        call  DEFINE_FIELD_LOCATIONS
      else if (CMATCH (key, 'M^ATERIAL')) then
        call  DEFINE_MATERIAL
      else if (CMATCH (key, 'P^RESTRESS'))   then
```

```
    call  DEFINE_PRESTRESS
  else if (CMATCH (key, 'SE^GMENTS')) then
    call  DEFINE_SEGMENTS
  else if (CMATCH (key, 'SY^MMETRY')) then
    call  DEFINE_SYMMETRY_CONDITIONS
  else
    print *, '*** Illegal or ambiguous keyword ', key,
 $           ' after DEFINE'
  end if
  return
  end
```

The program begins checking whether a keyword actually follows DEFINE. If so it compares them in the usual matter and calls appropriate input subroutines. These are described next.

7.1 A Digression: The Basics of Using GAL-DBM

Because the basic mechanics of writing (storing) and reading (loading) data are very similar throughout all the subroutines that load and store data, these basics are presented before we present the details of each subroutine. Thus, in the discussion we can focus on the important details within the basic outline presented here.

7.1.1 Storing GAL Data

The first step is to OPEN the GAL Library that is to contain the data to be stored; see §6. If the Library is already open, nothing needs to be done.

Next, each dataset needs to be installed in the Library. This is easily done with a call to GMPUNT, [2], §7.11. In this tutorial example the dataset is always installed (even if it already exists). So, if you store the same named data more than once during the same run, the old dataset will be marked as deleted and a new dataset of the same name will appear in the Library. In general this is a safe practice, because old data are still there until the Library is packed. Thus, old data can be retrieved by enabling a deleted dataset. (See *pack [3], §48.1 and *enable [3], §24.1.)

Then for each record of data we wish to store we must construct the record name, followed by writing the data. The record name (rname is used in the code) is constructed by subroutine GMCORN [2], §10.6. The data are written with a call to GMPUTN [2], §9.9.

In this tutorial example we have chosen the record key (name) and its associated record cycles (record group), [2], §5.1, to correspond to the array name and array indices used in the dimensioned arrays in the code. For example, if we have a one-dimensional array named number and we have used number(1) through number(12), we construct the record name to be number.1:12. Thus, number(3) in the code equals number.3 in the database. Here we have constructed a record group containing 12 records with each record containing one number. If you look at the record attributes (use the *rat directive [3], §49.4), you will see the record number has a low cycle = 1, a high cycle = 12, and a logical size = 1 (1 number).

Somewhat more complex is a two-dimensional array (a matrix). In this tutorial there is only one matrix, c, the system coefficient matrix. This matrix is square, n by n, so we choose the record name to be c.1:n. We have as many records as we have columns in the matrix. Thus, each record has a logical size equal to n, i.e., each record contains the n numbers for the column it represents. For example, c.5 contains the n numbers for column 5 of the matrix, c. See §8.0 for the implementation details.

The simpliest case is the record that contains only one value. For this case the record name does not have to contain any group cycle numbers; see [2], §5.1.

Finally, after the data for a dataset are stored in the GAL Library, the GAL-DBM buffers should be flushed. This insures that all the data are actually written to the GAL Library (file). Thus, if the next thing that happens causes your run to fail, you still have all the data properly stored up to that point.

In summary, to store data in an open GAL Library the following steps are needed:

1) Install the dataset — `GMPUNT`
2) Then for each record repeat these steps
   a) Construct the record name — `GMCORN`
   b) Store the data — `GMPUTN`
3) Finish with a buffer flush — `GMFLUB`

7.1.2 Loading GAL Data

Again, the first step is to `OPEN` the GAL Library that contains the data to be read; see §6. If the Library is already open, nothing needs to be done.

Next, for each dataset to be loaded the dataset sequence number must be determined for use in subsequent calls to other GAL subroutines. Given the dataset name the integer function `LMFIND` returns the dataset sequence number [2], §7.5.

Then for each record name in the dataset we must determine the record group cycles, so all the data in the records can be loaded. The low and high record group cycles (`ilow` and `ihigh`, respectively) are returned by the subroutine `GMCEGY` [2], §9.3.

Next, the record name is constructed so that all the records are loaded in one read operation. The record name is constructed by a call to `GMCORN` [2], §10.6.

Finally, the data are loaded from the GAL Library into a single variable or an array. The call to `GMGETN` loads numeric data [2], §9.5.

In summary, we have the following simple outline for loading data:

1) Find dataset sequence number — `LMFIND`
2) Then for each record repeat these steps
   a) Get record cycle information — `GMGECY`
   b) Construct record name — `GMCORN`
   c) Load the data — `GMGETN`

This is about as simple as it can be done for arrays. However, if only a single value record has been stored, it can be loaded with only steps 1 and 2c. The record name does not need to contain the cycle number(s) [2], §5.1.

A sophisticated Processor should do some more error checking. The sophisticated reader should be aware of the test for an error condition, LMERCD [2], §14.5. Calls to LMERCD are typically made after all reads (GMGETx [2], §9.5) and many times after all writes (GMPUTx [2], §9.9). Also, the subroutine GMGETx returns two arguments, n and m, that contain information about how much data have been read. These arguments can also be used for error checking.

In the code that follows in §7.2, the LOAD and STORE operations are code inline in the applicable subroutine. A higher level of abstraction can be used by writing cover subroutines that load and store data. These subroutines then call the appropriate GAL subroutines to load and store. This results in cleaner code, i.e., all database I/O is done in two subroutines; so a change of databases is easily accommodated. However, you do pay the price of a modest loss of efficiency because of the additional subroutine calls. The inline code is used here for tutorial purposes. Also for a small Processor, the additional abstraction is not really needed.

Now, lets look at the CLIP and GAL calls required to define the problem, either by interactively entering the data through commands then storing it, or by loading previously stored data.

7.2 Defining Segments

The `DEFINE SEGMENT` command *introduces* a series of segment-definition commands which are expected to have the form

```
SEGMENT = i    BEGIN = x_i^beg, y_i^beg    END = x_i^end, y_i^end
[ LOAD | STORE ]
```

where x_i^{beg}, y_i^{beg} are the x, y coordinates of the starting point of the i^{th} segment, and x_i^{end}, y_i^{end} are the x, y coordinates of the ending point. The segment list is terminated by an END command that takes the control back to the main program. In listing the coordinates, the following boundary traversal convention must be observed: a closed contour is traversed in the ***counterclockwise*** sense if the region of interest is outside the contour (a cavity problem), and in the ***clockwise*** sense if the region of interest is inside the contour (a finite body problem); see Figure 2-1, §2.

In the CLAMP metalanguage, the | says that one may specify either LOAD or STORE, but not both simultaneously. The specifications are shown in brackets, meaning that they may be omitted.

For example, to define and store a 4-segment boundary that encloses a square region whose corner points are (0,0), (4,0), (4,4) and (0,4), and which constitutes the region of interest, you say

```
DEFINE SEGMENTS
  SEG=1 BEGIN=0,0 END=0,4
  SEG=2 BEGIN=0,4 END=4,4
  SEG=3 BEGIN=4,4 END=4,0
  SEG=4 BEGIN=4,0 END=0,0
  STORE
  END
```

(Segments may be actually defined in any order; there is also no need to number them sequentially.)

The commands that enter the segment data, plus the LOAD, STORE and END command, are call ***subordinate commands***, because they can appear if and only if the command DEFINE SEGMENT has been entered. The DEFINE SEGMENT command, which introduces the subordinate commands, is said to be the *header command* (it also goes by the names *master command*, *parent command*, etc.).

The STORE command is optional; you don't have to store the data in the GAL Library, unless you wish to keep it for later use.

If you have already defined the segments and stored the data, you can use the following command sequence to load the segment data from the database.

```
                         DEFINE SEGMENTS
                           LOAD
                           END
```

The processing of the segment-definition commands is carried out within subroutine DEFINE SEGMENTS:

```
*
*    Read segment-definition data
*
      subroutine     DEFINE_SEGMENTS
*
      implicit       none
      include        'database.inc'
      include        'segment.inc'
      character*8    key, CCLVAL
      character*20   rname
      integer        iseg, n, mseg, ICLTYP, ICLVAL, ICLSEK
      integer        idsn, LMFIND, ilow, ihigh, nrec
      real           xy(2)
      logical        CMATCH
*
 1000   call      CLREAD  (' Segment data> ',
     $           ' Enter SEG=iseg BEG=xbeg,ybeg END=xend,yend&&'//
     $           ' or LOAD or STORE&&'//
     $           'Terminate with END')
*
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with SEG or END'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))           then
          return
        else if (CMATCH (key, 'S^EGMENT'))  then
          iseg =   ICLVAL(2)
          if (iseg .le. 0 .or. iseg .gt. MAXSEG)  then
            print *, '*** Segment number', iseg, ' out of range'
            go to 1000
          end if
          segdef(iseg) = 1
          if (numel(iseg) .le. 0)  numel(iseg) = 1
          if (ICLSEK(3, 'B^EGIN') .ne. 0)   then
            call   CLVALF (' ', 2, xy, n)
            if (n .ge. 1)   xbeg(iseg) = xy(1)
            if (n .ge. 2)   ybeg(iseg) = xy(2)
          end if
          if (ICLSEK(3, 'E^ND') .ne. 0)   then
            call   CLVALF (' ', 2, xy, n)
```

```
           if (n .ge. 1)   xend(iseg) = xy(1)
           if (n .ge. 2)   yend(iseg) = xy(2)
          end if
        else if (CMATCH (key, 'L^OAD'))  then
C     --- find dataset
          idsn = LMFIND ( ldi, 'SEGMENT ', 100 )
          if ( idsn .eq. 0 ) then
            print*, ' Cannot find SEGMENT dataset; nothing LOADed.'
            go to 1000
          end if
C     --- get record name cycles, construct record name & read data
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'SEGDEF ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'SEGDEF ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'I', segdef,
     $                   n, 0, 0, 0, 200 )
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'NUMEL ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'NUMEL ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'I', numel,
     $                   n, 0, 0, 0, 300 )
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'XBEG ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'XBEG ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', xbeg,
     $                   n, 0, 0, 0, 400 )
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'YBEG ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'YBEG ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', ybeg,
     $                   n, 0, 0, 0, 500 )
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'XEND ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'XEND ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', xend,
     $                   n, 0, 0, 0, 600 )
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'YEND ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'YEND ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', yend,
     $                   n, 0, 0, 0, 700 )
        else if (CMATCH (key, 'S^TORE')) then
```

```
C     --- install dataset
          call  GMPUNT ( ldi, 'SEGMENT ', idsn, 16, 1000 )
C     --- determine largest value of segdef
          do 100 n=MAXSEG,1,-1
            if ( segdef(n) .ne. 0 ) then
              mseg = n
              go to 200
            end if
 100      continue
 200      continue
C     --- construct record name & write data
          call  GMCORN ( rname, 'SEGDEF ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'I', segdef, mseg,
     $                 0, 0, 0, 1100 )
          call  GMCORN ( rname, 'NUMEL ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'I', numel, mseg,
     $                 0, 0, 0, 1200 )
          call  GMCORN ( rname, 'XBEG ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', xbeg, mseg,
     $                 0, 0, 0, 1300 )
          call  GMCORN ( rname, 'YBEG ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', ybeg, mseg,
     $                 0, 0, 0, 1400 )
          call  GMCORN ( rname, 'XEND ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', xend, mseg,
     $                 0, 0, 0, 1500 )
          call  GMCORN ( rname, 'YEND ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', yend, mseg,
     $                 0, 0, 0, 1600 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *, '*** Illegal keyword ', key,' in segment data'
        end if
        go to 1000
      end
```

The structure of this subroutine is typical of those that handle subordinate commands. A "do forever" construction is headed by a CLREAD call, and the loop is escaped only when an END command is detected. Notice the different prompt and verbose prompt input arguments.

This subroutine provides an example of the use of the "search for keyword" function ICLSEK described in [1], §5.2. A keyword match is followed by a value pair retrieval through the list-loading subroutine CLVALF described in [1], §7.2.

Note the careful handling of the case in which less than two values appear after either BEGIN or END. This facilitates *table editing*. For example, the command

```
S=3 B=45.2
```

resets XBEG(3) to 45.2; nothing else changes.

To load the data the outline presented in §7.1.2 is followed exactly. Note that, the variable n is set to MAXSEG before every call to GMGETN. This insures that no more than MAXSEG values are read into the arrays that are dimensioned to MAXSEG. The value of n is reset after each call to GMGETN because n returns the actual number of values read. A real production Processor would perform some data checking to make sure the number of values read for each record are the same. Also, real professionals would use LMERCD [2], §14.5, to check for various errors that may have occurred during the read.

To store the data that has been entered the outline presented in §7.1.1 is followed exactly. Note that, the high cycle for the records is determined by computing the largest index of **segdef** that contains a non-zero value. Thus, when this data is read later the number of segments defined is known from the high cycle number for the SEGMENT dataset records.

7.2.1 Digression on Subordinate Commands

Why have we used subordinate commands rather than making the user type the segment in the DEFINE command itself? Well, contrast the above definition of the square region with the following one:

```
DEFINE SEGMENT=1 BEGIN=0,0 END=4,0
DEFINE SEGMENT=2 BEGIN=4,0 END=4,4
DEFINE SEGMENT=3 BEGIN=4,4 END=0,4
DEFINE SEGMENT=4 BEGIN=0,4 END=0,0
```

This is not too different in terms of typing effort, so the decision for adopting a one-level and a two-level structure in terms of number of keystrokes is marginal. But note that by going to a two-level scheme we have effectively separated the action of *selecting what to define*, namely segments, from the *actual definition* by entering coordinate values. This is a key aspect of *object-oriented programming:* first *select*, then *operate*. Let us make this a command design principle:

$$\boxed{\textit{Try to separate selection from operation}}$$

If you are entering commands from a keyboard perhaps the advantages are not immediately apparent. But if you go to some form of interactive graphics input the advantages will be evident when you try to "cover" the commands through message-sending techniques. The user of such graphic system will then see SEGMENTS in a "model definition" menu, and by pointing to it he or she is transported to another screen or window in which the process of entering the segments is actually carried out.

7.3 Defining Elements

By default, each segment contains only one boundary element (see logic of DEFINE SEGMENT). To put more elements per segment you use the DEFINE ELEMENTS command. This introduces subordinate commands of the form

```
SEGMENT = i  ELEMENTS = n
[ LOAD | STORE ]
```

where n is the number of boundary elements in the i^{th} segment. The data is terminated by an END command. For the square region used as an example, let's say we want 10 BEs on segments 1 and 3, 15 BEs on segments 2 and 4, and store this data:

```
DEFINE ELEMENTS
  SEG=1 EL=10 ; SEG=3 EL=10 ; SEG=2 EL=15 ; SEG=4 EL=15
  STORE ; END
```

which illustrates the fact that data may be entered in any order. The implementation shown below actually allows a more general command form:

```
SEGMENTS = i1,...,ik  ELEMENTS = n1,...,nk
```

so that segment i_1 gets n_1 elements, segment i_2 gets n_2, and so on. The example above can be abbreviated to

```
DEFINE ELEMENTS
  SEG=1:4 EL=10,15,10,15
  STORE ; END
```

For this simple Processor, using a command like this is probably overkill. It is implemented in that fashion only to illustrate the processing of variable length integer lists via CLVALI [1], §7.2:

```
C
C    Define number of (equally spaced) boundary elements per segment
C
      subroutine    DEFINE_ELEMENTS
C
      implicit      none
      include       'database.inc'
      include       'segment.inc'
C
      character*4   key, CCLVAL
      character*20  rname
      integer       i, iseg, n, nseg
      integer       iseglist(MAXSEG), numelist(MAXSEG)
      integer       ICLTYP, ICLSEK
      integer       idsn, LMFIND, ilow, ihigh, nrec
      real          FCLVAL
      logical       CMATCH
```

```
*
 1000   call      CLREAD  (' Element data> ',
     $     ' Enter SEG = i1 ... ik  EL = ne1, ... nek&&'//
     $         ' or LOAD or STORE&&'//
     $         'Terminate with END')
*
        if (ICLTYP(1) .le. 0)           then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
           return
        else if (CMATCH (key, 'S^EG'))   then
          call  CLVALI (' ', -MAXSEG, iseglist, nseg)
          if (ICLSEK(0,'E^LEM') .eq. 0)  then
            print *, '*** Keyword ELEMENTS is missing'
            go to 1000
          end if
          call     CLVALI (' ', -MAXSEG, numelist, n)
          do 2500  i = 1,nseg
            iseg =  iseglist(i)
            if (iseg .le. 0 .or. iseg .gt. MAXSEG) then
              print *, '*** Segment number',iseg,' out of range'
            else
              numel(iseg) = max(numelist(i),1)
            end if
 2500       continue
        else if (CMATCH (key, 'L^OAD'))  then
C     --- find dataset
          idsn = LMFIND ( ldi, 'SEGMENT ', 100 )
          if ( idsn .eq. 0 ) then
            print*, ' Cannot find SEGMENT dataset; nothing LOADed.'
            go to 1000
          end if
C     --- get record name cycles, construct record name & read data
          n = MAXSEG
          call  GMGECY ( ' ', ldi, idsn, 'NUMEL ', nrec, ilow,
     $                   ihigh, 200 )
          call  GMCORN ( rname, 'NUMEL ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'I', numel,
     $                   n, 0, 0, 0, 300 )
        else if (CMATCH (key, 'S^TORE')) then
C     --- find dataset because this is an update of SEGMENT_NUMEL
          idsn = LMFIND ( ldi, 'SEGMENT ', 500 )
          if ( idsn .eq. 0 ) then
            print*, ' Cannot find SEGMENT dataset; nothing STOREd.'
            print*, ' Must DEFINE SEGMENTs before DEFINing ELEMENTs.'
            go to 1000
          end if
C     --- construct record name & write data
          call  GMGECY ( ' ', ldi, idsn, 'NUMEL ', nrec, ilow,
     $                  ihigh, 200 )
          call  GMCORN ( rname, 'NUMEL ', 1, nrec )
          call  GMPUTN ( 'W/U', ldi, idsn, rname, 'I', numel, nrec,
     $                  0, 0, 0, 1200 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *, '*** Illegal keyword ', key,' in element data'
        end if
        go to 1000
      end
```

Here the data are loaded following the outline given in §7.1.2. However, note that, the NUMEL data are also loaded under DEFINE SEGMENTS, so it is not necessary to reload the data here. As stated above the DEFINE ELEMENTS command is an overkill, so we end up with this strange construction. The first author recommends including the number of element definitions under DEFINE SEGMENTS as a subordinate command that calls DEFINE.ELEMENTS, then all segment data operations are encapsulated in the same place. See §7.3.1 below for the second author's opinion.

Thus, to store the data here we must be sure the SEGMENT dataset exists instead of the usual install operation (the dataset is installed in the DEFINE SEGMENTS code, §7.2). If the dataset is found we proceed, but not along the standard path. First, since the SEGMENT dataset and the NUMEL record already exist we retrieve the record cycles by calling GMGECY [2], §9.3. Then, the record name is constructed with a call to GMCORN [2], §10.6. Finally, note that, the op_code, the first argument, in GMPUTN is set to write/update. That is, we write over the existing data. See [2], §9.9 for more information on the op_codes.

If you can't follow the code, don't worry. It is more advanced than the typical input routine in DBEM2, so you can study it later.

7.3.1 Digression: Simplifying Commands

Why didn't we allow element data to be specified in the same commands that define the segment geometry? For example, we might have allowed commands such as

```
SEG = 13  BEG = -1.50,3.53  END = 14.81,6.22  ELEM = 5
```

The answer fits within another design principle:

$$\boxed{\textit{Keep commands simple}}$$

Simplicity is an admirable general principle, but for our case something more specific applies:

Don't mix persistent and volatile data in the same command

The terms "persistent" and "volatile" are used in a relative sense to denote degrees of "changeability" of the data. For example, segment data are more persistent than element data, since presumably you want to solve a problem whose geometry is dictated by external requirements; typically by engineering considerations. On the other hand, the number of elements per segment is a judgement decision: the program user attempts to get satisfactory accuracy (more elements, more accuracy) with reasonable cost (more elements, more computer time).

Frequently the number of elements is varied while keeping the segment data fixed; this is called a *convergence study*. So there are good reasons to separate the commands that define these two aspects.

7.4 Defining Boundary Conditions

Each segment may be given a different boundary condition (BC) that involves any of the following stress/displacement combinations:

BC Code	*Prescribed boundary values*
0	Shear stress σ_s and normal stress σ_n
1	Shear displacement u_s and normal displacement u_n
2	Shear displacement u_s and normal stress σ_n
3	Shear stress σ_s and normal displacement u_n

These values are *constant* along the segment, so they can be read on a segment-by-segment basis. The stress values are understood to be *resultants* over the segment.

(The "BC codes" are related to those used by Crouch and Starfield [4]. Using integer codes is far from the best way to implement readable software, but we shall follow their convention.)

The BC data commands are introduced by a `DEFINE BOUNDARY_CONDITIONS` header command (which may be abbreviated to just `D B`), and have the form

$$\texttt{SEG = } i \quad \{\texttt{SS = } \sigma_s \;\mid\; \texttt{SD = } u_s\} \quad \{\texttt{NS = } \sigma_n \;\mid\; \texttt{ND = } u_n\}$$
$$\texttt{[LOAD | STORE]}$$

terminated by an `END` command. Keyword `SS` means shear stress, `SD` shear displacement, and so on.

In the CLAMP metalanguage, the | indicates that one may specify either σ_s or u_s, but not both simultaneously, and similarly for σ_n and u_n. The specifications are shown in braces, meaning that they may not be omitted.

If no BC is ever specified for segment i, that segment is assumed stress free (code 0 with $\sigma_s = \sigma_n = 0$). If only a normal value is prescribed, a zero shear stress is assumed, and so on.

The implementation of `DEFINE_BOUNDARY` follows.

```
*
*    Read boundary condition data for segments
*
      subroutine     DEFINE_BOUNDARY_CONDITIONS
C
      implicit       none
      include        'database.inc'
      include        'segment.inc'
*
      character*4    key, CCLVAL, word(2)
      character*20   rname
      integer        iseg, n, nw, iloc(2)
      integer        ICLVAL, ICLSEK, ICLTYP
      integer        idsn, LMFIND, ilow, ihigh, nrec, mseg
      logical      CMATCH
*
 1000  call     CLREAD  (' Bound_cond data> ',
     $   ' Enter SEG=iseg {SS=sig_s | SD=u_s} {NS=sig_n | ND=u_n}'//
     $   ' or LOAD or STORE&&'//
     $    '&&Terminate with END')
*
       if (ICLTYP(1) .le. 0)            then
         print *, '*** Command must begin with keyword'
         go to 1000
       end if
       key =     CCLVAL(1)
       if (CMATCH (key, 'E^ND'))        then
         return
       else if (CMATCH (key, 'S^EG'))   then
         iseg =   ICLVAL(2)
         if (iseg .le. 0 .or. iseg .gt. MAXSEG)  then
           print *, '*** Segment number', iseg, ' is out of range'
           go to 1000
         end if
         call  CLOADK ('L', -2, word, iloc, nw)
         call  BCVALUES  (iseg, nw, word, iloc)
       else if (CMATCH (key, 'L^OAD'))  then
C     --- find dataset
         idsn = LMFIND ( ldi, 'BCVALUES ', 100 )
         if ( idsn .eq. 0 ) then
           print*, ' Cannot find BCVALUES dataset; nothing LOADed.'
           go to 1000
         end if
C     --- get record name cycles, construct record name & read data
         n = MAXSEG
         call  GMGECY ( ' ', ldi, idsn, 'KODE ', nrec, ilow,
     $                  ihigh, 220 )
         call  GMCORN ( rname, 'KODE ', ilow, ihigh )
         call  GMGETN ( 'R/L', ldi, idsn, rname, 'I', kode,
     $                  n, 0, 0, 0, 200 )
         n = MAXSEG
         call  GMGECY ( ' ', ldi, idsn, 'BVN ', nrec, ilow,
     $                  ihigh, 220 )
         call  GMCORN ( rname, 'BVN ', ilow, ihigh )
         call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', bvn,
     $                  n, 0, 0, 0, 300 )
         n = MAXSEG
         call  GMGECY ( ' ', ldi, idsn, 'BVS ', nrec, ilow,
     $                  ihigh, 220 )
         call  GMCORN ( rname, 'BVS ', ilow, ihigh )
         call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', bvs,
     $                  n, 0, 0, 0, 400 )
```

```
        else if (CMATCH (key, 'S^TORE')) then
C     --- install dataset
          call  GMPUNT ( ldi, 'BCVALUES ', idsn, 16, 1000 )
C     --- determine largest index of data stored
          do 100 n=MAXSEG,1,-1
            if ( (kode(n) .ne. 0) .or. (bvn(n) .ne. 0.0)
     $           .or. (bvs(n) .ne. 0.0) ) then
              mseg = n
              go to 200
            end if
  100     continue
  200     continue
C     --- construct record name & write data
          call  GMCORN ( rname, 'KODE ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'I', kode, mseg,
     $                   0, 0, 0, 1100 )
          call  GMCORN ( rname, 'BVN ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', bvn, mseg,
     $                   0, 0, 0, 1200 )
          call  GMCORN ( rname, 'BVS ', 1, mseg )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', bvs, mseg,
     $                   0, 0, 0, 1300 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *, '*** Illegal keyword ', key, ' in BC data'
        end if
        go to 1000
      end
```

This subroutine follows the outline for LOAD and STORE given in §7.1.2 and §7.1.1. This also illustrates the use of the "load keyword" entry points of [1], §8.2. These calls search for keywords such as SS and move them to the subroutine work area. This simplifies keyword legality tests such as "SS and SD cannot appear in the same command." To do these chores DEFINE_BOUNDARY calls subroutine BCVALUES:

```
*
*    Store boundary condition values in tables
*
      subroutine    BCVALUES
     $             (iseg, nw, word, iloc)
*
      implicit      none
      include       'segment.inc'
      character*(*) word(2)
      real          FCLVAL
      integer       iseg, nw, iloc(2)
      integer       code, i, isd, iloads, iloadn, ks, kd, kn
      logical       CMATCH
*
```

```
      ks =    0
      kn =    0
      kd =    0
      isd =   0
      iloadn =  0
      iloads =  0
*
      do 2000  i = 1,nw
        if (CMATCH (word(i), 'SS'))  then
          ks =   ks + 1
          iloads = iloc(i)
        else if (CMATCH (word(i), 'SD'))  then
          ks =   ks + 1
          kd =   kd + 1
          isd =  1
          iloads = iloc(i)
        else if (CMATCH (word(i), 'NS'))  then
          kn =   kn + 1
          iloadn = iloc(i)
        else if (CMATCH (word(i), 'ND'))  then
          kn =   kn + 1
          kd =   kd + 1
          iloadn = iloc(i)
        else
          print *, '*** Illegal BC keyword ', word(i),'  segment',iseg
          return
        end if
        if (kn .gt. 1 .or. ks .gt. 1)    then
          print *, '*** Illegal BC combination for segment', iseg
          return
        end if
2000  continue
*
      if (iloadn .gt. 0)         bvn(iseg) =  FCLVAL(iloadn+1)
      if (iloads .gt. 0)         bvs(iseg) =  FCLVAL(iloads+1)
*
      if (kd .eq. 0)             then
        code =  1
      else if (kd .eq. 1)        then
        code =  3
        if (isd .eq. 0)          code = 4
      else
        code =  2
      end if
      kode(iseg) =  code-1
      return
      end
```

which embodies the logic for eventually storing the user-supplied values into appropriate spots in arrays BVS and BVN.

7.5 Defining Symmetry Conditions

If the problem exhibits symmetry conditions, commands to specify symmetry axes are introduced by the header command DEFINE SYMMETRY_CONDITIONS (which may be abbreviated to just D S) and have the form

```
XSYM = $x_{sym}$
YSYM = $y_{sym}$
[ LOAD | STORE ]
```

terminated by an END command. The XSYM command specifies that $x = x_{sym}$ is a line of symmetry parallel to the x axis. The YSYM command specifies that $y = y_{sym}$ is a line of symmetry parallel to the y axis. One or two specifications may be given. The Processor logic does not allow "skew" symmetry conditions.

The implementation of the DEFINE_SYMMETRY routine is straightforward:

```
*
*     Read symmetry condition data
*
      subroutine   DEFINE_SYMMETRY_CONDITIONS
*
      implicit       none
      include        'database.inc'
      include        'symmetry.inc'
      character*4    key, CCLVAL
      integer        ixsym, iysym, ICLTYP, idsn, LMFIND, n
      real           FCLVAL
      logical        CMATCH
*
      ixsym =  mod(ksym,2)
      iysym =  ksym/2
*
 1000   call       CLREAD  (' Symmetry data> ',
     $     ' Enter  XSYM=xsym  or  YSYM=ysym or LOAD or STORE '//
     $      '&&Terminate with END')
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
*
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))       then
          return
        else if (CMATCH (key, 'X^SYM')) then
          xsym =    FCLVAL(2)
          ixsym =  1
          ksym =  2*iysym + ixsym
        else if (CMATCH (key, 'Y^SYM')) then
```

```
          ysym =   FCLVAL(2)
          iysym =  1
          ksym =  2*iysym + ixsym
        else if (CMATCH (key, 'L^OAD'))  then
C     --- find dataset
          idsn = LMFIND ( ldi, 'SYMMETRY ', 100 )
          if ( idsn .eq. 0 ) then
            print*, ' Cannot find SYMMETRY dataset; nothing LOADed.'
            go to 1000
          end if
C     --- read data
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'KSYM ', 'I', ksym,
     $                   n, 0, 0, 0, 200 )
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'XSYM ', 'I', xsym,
     $                   n, 0, 0, 0, 300 )
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'YSYM ', 'I', ysym,
     $                   n, 0, 0, 0, 400 )
        else if (CMATCH (key, 'S^TORE')) then
C     --- install dataset
          call  GMPUNT ( ldi, 'SYMMETRY ', idsn, 16, 500 )
C     --- write data
          call  GMPUTN ( 'W', ldi, idsn, 'KSYM ', 'I', ksym, 1,
     $                   0, 0, 0, 600 )
          call  GMPUTN ( 'W', ldi, idsn, 'XSYM ', 'I', xsym, 1,
     $                   0, 0, 0, 700 )
          call  GMPUTN ( 'W', ldi, idsn, 'YSYM ', 'I', ysym, 1,
     $                   0, 0, 0, 800 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *,  '*** Illegal keyword ', key,' in symmetry data'
        end if
        go to 1000
      end
```

(Here KSYM is an integer "symmetry flag" related to that used in the original TWOBI program.)

The LOAD and STORE commands are implemented following the outline given in §7.1.2 and §7.1.1. However, here each record is just one number, so we do not have to construct the record name. For more information see [2], §5.1 for the details of record naming.

7.6 Defining Material Properties

Material properties are introduced by a DEFINE MATERIAL header command (which can be abbreviated to just D M). The commands have a simple form:

```
EM = E
PR = ν
[ LOAD | STORE ]
```

terminated by an END command. The EM command specifies the elastic modulus and the PR command specifies Poisson's ratio. Since DBEM2 is restricted to elastic isotropic materials and does not consider thermal effects, these two material properties suffice.

The default values for E and ν set by CLEAR are 1.0 and 0.0, respectively.

The implementation of DEFINE MATERIALS is straightforward and does not involve any fancy new construct:

```
*
*     Read material property data
*
      subroutine    DEFINE_MATERIAL
*
      implicit      none
      include       'database.inc'
      include       'material.inc'
      character*4   key, CCLVAL
      integer       ICLTYP, idsn, LMFIND, n
      real          FCLVAL
      logical       CMATCH
*
 1000   call      CLREAD  (' Material data> ',
     $            ' Enter EM=em  or  PR=pr or LOAD or STORE&&'//
     $            'Terminate with END')
*
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
          return
        else if (CMATCH (key, 'EM'))     then
          em =     FCLVAL(0)
        else if (CMATCH (key, 'P^R'))    then
          pr =     FCLVAL(0)
        else if (CMATCH (key, 'L^OAD'))  then
C     --- find dataset
          idsn = LMFIND ( ldi, 'MATERIAL ', 100 )
          if ( idsn .eq. 0 ) then
```

```
            print*, ' Cannot find MATERIAL dataset; nothing LOADed'
            go to 1000
          end if
C     --- read data
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'EM ', 'S', em, n, 0,
     $                 0, 0, 200 )
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'PR ', 'S', pr, n, 0,
     $                 0, 0, 300 )
        else if (CMATCH (key, 'S^TORE')) then
C     --- install dataset
          call  GMPUNT ( ldi, 'MATERIAL ', idsn, 16, 500 )
C     --- write data
          call  GMPUTN ( 'W', ldi, idsn, 'EM ', 'S', em, 1, 0, 0,
     $                 0, 600 )
          call  GMPUTN ( 'W', ldi, idsn, 'PR ', 'S', pr, 1, 0, 0,
     $                 0, 700 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *, '*** Illegal keyword ', key,' in material data'
        end if
        go to 1000
      end
```

Here the LOAD and STORE are identical to the previous implementation.

7.7 Defining Prestress Data

If the initial stress state has nonzero components, prestress data have to be introduced by a DEFINE PRESTRESS header. The prestress-definition commands have a very simple form:

SXXO = σ^0_{xx}
SYYO = σ^0_{yy}
SXYO = σ^0_{xy}
[LOAD | STORE]

As usual, these commands are terminated by an END command. Undefined prestress components are assumed zero.

The implementation of DEFINE PRESTRESS is quite similar to that of DEFINE_MATERIAL :

```
*
*     Read prestress (initial field stresses) data
*
      subroutine    DEFINE_PRESTRESS
*
      implicit      none
      include       'database.inc'
      include       'prestress.inc'
      character*4   key, CCLVAL
      integer       ICLTYP, idsn, LMFIND, n
      real          FCLVAL
      logical       CMATCH
*
 1000  call       CLREAD  (' Prestress data> ',
     $             ' Enter SXXO=sxx0, SYYO=syy0 or SXYO=sxy0&&'//
     $             ' or LOAD or STORE&&'//
     $             'Terminate with END')
*
        if (ICLTYP(1) .le. 0)           then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
          return
        else if (CMATCH (key, 'SX^XO'))     then
          sxx0 =  FCLVAL(0)
        else if (CMATCH (key, 'SY^YO'))    then
          syy0 =  FCLVAL(0)
        else if (CMATCH (key, 'SX^YO'))    then
          sxy0 =  FCLVAL(0)
        else if (CMATCH (key, 'L^OAD'))  then
C     --- find dataset
          idsn = LMFIND ( ldi, 'PRESTRESS ', 100 )
          if ( idsn .eq. 0 ) then
            print*, ' Cannot find PRESTRESS dataset; nothing LOADed.'
            go to 1000
          end if
C     --- read data
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'SXXO ', 'S', sxx0,
     $                   n, 0, 0, 0, 200 )
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'SYYO ', 'S', syy0,
     $                   n, 0, 0, 0, 300 )
          n = 1
          call  GMGETN ( 'R/L', ldi, idsn, 'SXYO ', 'S', sxy0,
     $                   n, 0, 0, 0, 400 )
        else if (CMATCH (key, 'S^TORE')) then
C     --- install dataset
          call  GMPUNT ( ldi, 'PRESTRESS ', idsn, 16, 500 )
C     --- write data
          call  GMPUTN ( 'W', ldi, idsn, 'SXXO ', 'S', sxx0, 1,
     $                   0, 0, 0, 600 )
          call  GMPUTN ( 'W', ldi, idsn, 'SYYO ', 'S', syy0, 1,
     $                   0, 0, 0, 700 )
          call  GMPUTN ( 'W', ldi, idsn, 'SXYO ', 'S', sxy0, 1,
     $                   0, 0, 0, 800 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *,'*** Illegal keyword ', key,' in prestress data'
        end if
        go to 1000
      end
```

7.8 Defining Output Field Locations

The last piece of input data is not related to the problem definition, but to the specification of the field points at which the program user would like to get computed results, *viz.*, displacements and stresses.

(This set of information is characteristic of boundary element methods, in which all basic givens and unknowns are at the boundary. If you want information at field points not on the boundary, you have to ask for it and specify where.)

For convenience the output locations are not specified point by point, but as equally spaced points on line segments. You specify the location of the first and last point on the line, and the number of points, if any, to be "collocated" between the first and last one.

The output field location specification commands are introduced by a `DEFINE FIELD-LOCATIONS` header command (which may be abbreviated to `D F`) and have a form reminiscent of the segment-definition commands:

LINE = i FIRST = x_i^{first}, y_i^{first} LAST = x_i^{last}, y_i^{last} [POINTS=n_{int}]
[LOAD | STORE]

terminated by an `END` command. Here n_{int} is the number of *intermediate* points to be inserted (equally spaced) between the first and last point. If this phrase is omitted, $n_{int} = 0$ is assumed so only the first and last points will be output points. If the first and last points coincide, output will be at only one point.

For example:

```
DEF OUT
  LINE=1 F=2@0.2 L=2@3.8 P=9
  LINE=2 F=3.8,0.2 L=0.2,3.8 P=9
  STORE
  END
```

specifies two output lines running at 45° and 135°, respectively, with 11 output points (first+last+9) in each, and the data are stored.

Here is the implementation of the `DEFINE OUTPUT LOCATIONS` routine:

```
C
C     Read location of output field points
C
      subroutine   DEFINE_FIELD_LOCATIONS
C
      implicit       none
      include        'database.inc'
      include        'output.inc'
      character*8    key, CCLVAL
      character*20   rname
      real           FCLVAL
      integer        ilin, n, mark, ICLVAL, ICLSEK, ICLTYP
      integer       idsn, LMFIND, ilow, ihigh, nrec, mlin
      real          xy(2)
      logical       onepoint, CMATCH
*
 1000   call      CLREAD  (' Field location data> ',
     $   ' Enter LIN=ilin FIRST=xfirst,yfirst LAST=xlast,ylast'//
     $    '[P=ninter]&& or LOAD or STORE&&Terminate with END')
*
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
          return
        else if (CMATCH (key, 'LI^NE'))   then
          ilin =   ICLVAL(2)
          if (ilin .le. 0 .or. ilin .gt. MAXLIN)  then
            print *, '*** Field line number',ilin,' is out of range'
            go to 1000
          end if
          lindef(ilin) = 1
          nintop(ilin) = 0
          onepoint = .true.
          if (ICLSEK(3, 'F^IRST') .ne. 0)  then
            call   CLVALF (' ', 2, xy, n)
            if (n .ge. 1)   xfirst(ilin) = xy(1)
            if (n .ge. 2)   yfirst(ilin) = xy(2)
          end if
          if (ICLSEK(3, 'L^AST') .ne. 0)   then
            call   CLVALF (' ', 2, xy, n)
            if (n .ge. 1)   xlast(ilin) = xy(1)
            if (n .ge. 2)   ylast(ilin) = xy(2)
            onepoint = .false.
          end if
          if (onepoint)     then
            xlast(ilin) =  xfirst(ilin)
            ylast(ilin) =  yfirst(ilin)
          end if
          if (ICLSEK(3, 'P^OINTS') .ne. 0) then
            nintop(ilin) = max(ICLVAL(0),0)
          end if
        else if (CMATCH (key, 'LO^AD'))  then
C      --- find dataset
          idsn = LMFIND ( ldi, 'FIELD ', 100 )
          if ( idsn .eq. 0 ) then
            print*, ' Cannot find FIELD dataset; nothing LOADed.'
            go to 1000
          end if
C     --- get record name cycles, construct record name & read data
          n = MAXLIN
          call  GMGECY ( ' ', ldi, idsn, 'LINDEF ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'LINDEF ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'I', lindef,
     $                   n, 0, 0, 0, 200 )
          n = MAXLIN
          call  GMGECY ( ' ', ldi, idsn, 'NINTOP ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'NINTOP ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'I', nintop,
     $                   n, 0, 0, 0, 300 )
          n = MAXLIN
          call  GMGECY ( ' ', ldi, idsn, 'XFIRST ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'XFIRST ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', xfirst,
     $                   n, 0, 0, 0, 400 )
          n = MAXLIN
          call  GMGECY ( ' ', ldi, idsn, 'YFIRST ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'YFIRST ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', yfirst,
     $                   n, 0, 0, 0, 500 )
          n = MAXLIN
          call  GMGECY ( ' ', ldi, idsn, 'XLAST ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'XLAST ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', xlast,
     $                   n, 0, 0, 0, 600 )
          n = MAXLIN
          call  GMGECY ( ' ', ldi, idsn, 'YLAST ', nrec, ilow,
     $                   ihigh, 220 )
          call  GMCORN ( rname, 'YLAST ', ilow, ihigh )
          call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', ylast,
     $                   n, 0, 0, 0, 700 )
        else if (CMATCH (key, 'S^TORE')) then
C     --- install dataset
          call  GMPUNT ( ldi, 'FIELD ', idsn, 16, 1000 )
C     --- determine largest value of lindef
          do 100 n=MAXLIN,1,-1
            if ( lindef(n) .ne. 0 ) then
              mlin = n
              go to 200
            end if
  100     continue
  200     continue
C     --- construct record name & write data
          call  GMCORN ( rname, 'LINDEF ', 1, mlin )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'I', lindef, mlin,
     $                  0, 0, 0, 1100 )
          call  GMCORN ( rname, 'NINTOP ', 1, mlin )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'I', nintop, mlin,
     $                  0, 0, 0, 1200 )
          call  GMCORN ( rname, 'XFIRST ', 1, mlin )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', xfirst, mlin,
     $                  0, 0, 0, 1300 )
          call  GMCORN ( rname, 'YFIRST ', 1, mlin )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', yfirst, mlin,
     $                  0, 0, 0, 1400 )
          call  GMCORN ( rname, 'XLAST ', 1, mlin )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', xlast, mlin,
     $                  0, 0, 0, 1500 )
          call  GMCORN ( rname, 'YLAST ', 1, mlin )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', ylast, mlin,
     $                  0, 0, 0, 1600 )
          call  GMFLUB ( ldi, 0, 2000 )
        else
          print *, '*** Illegal keyword ', key,' in field loc data'
        end if
        go to 1000
      end
```

Once again, the LOAD and STORE command implementations follow the outline given in §7.1.2 and §7.1.1.

The input data section is complete.

THIS PAGE LEFT BLANK INTENTIONALLY.

8. Solving the Problem

Having finished input data preparation, the three steps involved in solving the elasto-static problem are as follows.

Building the Boundary Element Model. The input data have defined the geometry of the problem in terms of segments. Segments are broken down into equally spaced boundary elements. The first step consists of building element-by-element data, and is carried out when you enter the command BUILD.

Assembling the Discrete Equations. This step generates a matrix **C** of "influence coefficients" and a vector **r** of "forcing functions." These arrays have dimensions equal to twice the total number of boundary elements. The construction of the elements of **C** and **r** follows the direct formulation of boundary-integral methods and is not explained here. This step is triggered by the command GENERATE and is carried out by subroutine GENERATE and subordinate routines.

Solving for the unknowns. The linear equation system **Cx** = **r** is solved (by a Gauss elimination method) for vector **x**, which contains the boundary unknowns. This step is triggered by command SOLVE and is carried out by subroutine SOLVE and a subordinate routine.

For each of these commands, BUILD, GENERATE, and SOLVE the qualifiers LOAD and STORE can be used. If the qualifier is LOAD then the data are loaded from the open GAL Library without computing the data. If the qualifier is STORE the data are computed, then stored in the GAL Library. The presence of the qualifier is determined in the subroutine DO_COMMAND; see §5. The value of the qualifier is passed to BUILD, GENERATE, and SOLVE as the argument op, a blank value is the default.

All of the GAL entry points and methods for loading and storing data have been illustrated previously in §7.0, and since we are not going to explain the theory behind these tasks, the BUILD, GENERATE and SOLVE subroutines are listed next without commentary.

```
*
*     Build detailed boundary element data
*
      subroutine  BUILD ( op )
*
      implicit       none
      include        'database.inc'
      include        'segment.inc'
      include        'element.inc'
      include        'material.inc'
      include        'prestress.inc'
      character*(*)   op
      character*20    rname
      integer         iseg, k, ne, num
      integer         idsn, LMFIND, ilow, ihigh, nrec, n
      real            xd, yd, side
*
```

```
      if ( op .eq. 'LOAD') go to 5000
      k =   0
      do 2000  iseg  = 1,MAXSEG
        if (segdef(iseg) .eq. 0)   go to 2000
        num =  numel(iseg)
        xd =   (xend(iseg)-xbeg(iseg))/num
        yd =   (yend(iseg)-ybeg(iseg))/num
        side = sqrt(xd**2+yd**2)
        if (side .eq. 0.0)         go to 2000
        do 1500  ne = 1,num
          k =   k + 1
          if (k .gt. MAXELM)     then
            print *, '*** Boundary element count exceeds ',MAXELM
            print *, '    Excess elements ignored'
            return
          end if
          xme(k) =  xbeg(iseg) + 0.5*(2.*ne-1)*xd
          yme(k) =  ybeg(iseg) + 0.5*(2.*ne-1)*yd
          hleng(k) = 0.5*side
          sinbet(k) = yd/side
          cosbet(k) = xd/side
          b(2*k-1) = bvs(iseg)
          b(2*k  ) = bvn(iseg)
          kod(k) =  kode(iseg)
 1500     continue
 2000   continue
      numbe =  k
      print '('' Discrete model building completed:'',
     $         I5,'' boundary elements''/)', numbe
      if ( op .eq. 'STORE' ) then
C   --- STORE data
C   --- install dataset
        call  GMPUNT ( ldi, 'ELEMENT ', idsn, 16, 1000 )
C   --- construct record name & write data
        call  GMPUTN ( 'W', ldi, idsn, 'NUMBE ', 'I', numbe,
     $                 1, 0, 0, 0, 1050 )
        call  GMCORN ( rname, 'XME ', 1, numbe )
        call  GMPUTN ( 'W', ldi, idsn, rname, 'S', xme, numbe,
     $                 0, 0, 0, 1100 )
        call  GMCORN ( rname, 'YME ', 1, numbe )
        call  GMPUTN ( 'W', ldi, idsn, rname, 'S', yme, numbe,
     $                 0, 0, 0, 1200 )
        call  GMCORN ( rname, 'HLENG ', 1, numbe )
        call  GMPUTN ( 'W', ldi, idsn, rname, 'S', hleng, numbe,
     $                 0, 0, 0, 1300 )
        call  GMCORN ( rname, 'SINBET ', 1, numbe )
        call  GMPUTN ( 'W', ldi, idsn, rname, 'S', sinbet, numbe,
     $                 0, 0, 0, 1400 )
```

```
          call   GMCORN ( rname, 'COSBET ', 1, numbe )
          call   GMPUTN ( 'W', ldi, idsn, rname, 'S', cosbet, numbe,
     $                    0, 0, 0, 1500 )
          call   GMCORN ( rname, 'KOD ', 1, numbe )
          call   GMPUTN ( 'W', ldi, idsn, rname, 'I', kod, numbe,
     $                    0, 0, 0, 1600 )
          call   GMCORN ( rname, 'B ', 1, 2*numbe )
          call   GMPUTN ( 'W', ldi, idsn, rname, 'S', b, 2*numbe,
     $                    0, 0, 0, 1500 )
          call   GMFLUB ( ldi, 0, 2000 )
        end if
        return
*
 5000 continue
C --- LOAD data
C --- find dataset
        idsn = LMFIND ( ldi, 'ELEMENT ', 100 )
        if ( idsn .eq. 0 ) then
          print*, ' Cannot find ELEMENT dataset; nothing LOADed.'
          return
        end if
C --- get record name cycles, construct record name & read data
        n = 1
        call   GMGETN ( 'R/L', ldi, idsn, 'NUMBE ', 'I', numbe,
     $                  n, 0, 0, 0, 150 )
        n = MAXELM
        call   GMGECY ( ' ', ldi, idsn, 'XME ', nrec, ilow,
     $                  ihigh, 180 )
        call   GMCORN ( rname, 'XME ', ilow, ihigh )
        call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', xme,
     $                  n, 0, 0, 0, 200 )
        n = MAXELM
        call   GMGECY ( ' ', ldi, idsn, 'YME ', nrec, ilow,
     $                  ihigh, 280 )
        call   GMCORN ( rname, 'YME ', ilow, ihigh )
        call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', yme,
     $                  n, 0, 0, 0, 300 )
        n = MAXELM
        call   GMGECY ( ' ', ldi, idsn, 'HLENG ', nrec, ilow,
     $                  ihigh, 380 )
        call   GMCORN ( rname, 'HLENG ', ilow, ihigh )
        call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', hleng,
     $                  n, 0, 0, 0, 400 )
        n = MAXELM
        call   GMGECY ( ' ', ldi, idsn, 'SINBET ', nrec, ilow,
     $                  ihigh, 480 )
        call   GMCORN ( rname, 'SINBET ', ilow, ihigh )
        call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', sinbet,
$                n, 0, 0, 0, 500 )
      n = MAXELM
      call   GMGECY ( ' ', ldi, idsn, 'COSBET ', nrec, ilow,
$                ihigh, 580 )
      call   GMCORN ( rname, 'COSBET ', ilow, ihigh )
      call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', cosbet,
$                n, 0, 0, 0, 600 )
      n = MAXELM
      call   GMGECY ( ' ', ldi, idsn, 'KOD ', nrec, ilow,
$                ihigh, 680 )
      call   GMCORN ( rname, 'KOD ', ilow, ihigh )
      call   GMGETN ( 'R/L', ldi, idsn, rname, 'I', kod,
$                n, 0, 0, 0, 700 )
      n = MAXEQS
      call   GMGECY ( ' ', ldi, idsn, 'B ', nrec, ilow,
$                ihigh, 780 )
      call   GMCORN ( rname, 'B ', ilow, ihigh )
      call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', b,
$                n, 0, 0, 0, 800 )
*
      return
      end
```

```
*
*     Calculate influence coefficient matrix and RHS vector
*
      subroutine  GENERATE ( op )
*
      implicit        none
      include        'database.inc'
      include        'material.inc'
      include        'element.inc'
      include        'prestress.inc'
      include        'symmetry.inc'
      character*(*)   op
      character*20    rname
      integer         i, j, n, nrec, ilow, ihigh, LMFIND, idsn, igap
      real            sinbi, cosbi, sinbj, cosbj, ss0, sn0, g
      real            xi, xj, yi, yj, sj
      real            ass, asn, ans, ann, bss, bsn, bns, bnn
*
      if ( op .eq. 'LOAD' )  go to 4000
      g =    0.5*em/(1.+pr)
      do 3000  i = 1,numbe
        r(2*i-1) = 0.
        r(2*i  ) = 0.
        xi =  xme(i)
        yi =  yme(i)
        cosbi =  cosbet(i)
```

```
          sinbi =  sinbet(i)
          do 2500  j = 1,numbe
            ass =  0.0
            asn =  0.0
            ans =  0.0
            ann =  0.0
            bss =  0.0
            bsn =  0.0
            bns =  0.0
            bnn =  0.0
            xj =   xme(j)
            yj =   yme(j)
            cosbj =  cosbet(j)
            sinbj =  sinbet(j)
            sj =     hleng(j)
            ss0 = (syy0-sxx0)*sinbj*cosbj + sxy0*(cosbj**2-sinbj**2)
            sn0 = sxx0*sinbj**2 - 2.*sxy0*sinbj*cosbj + syy0*cosbj**2
            call   COEFF (xi, yi, xj, yj, sj,
     $                  1, em, pr, cosbi, sinbi, cosbj, sinbj,
     $                  ass, asn, ans, ann, bss, bsn, bns, bnn)
            if (ksym .eq. 1 .or. ksym .eq. 3)  then
              call   COEFF (xi, yi, 2.*xsym-xme(j), yj, sj,
     $                    -1, em, pr, cosbi, sinbi, cosbj, -sinbj,
     $                    ass, asn, ans, ann, bss, bsn, bns, bnn)
            end if
            if (ksym .eq. 2 .or. ksym .eq. 3)  then
              call   COEFF (xi, yi, xj, 2.*ysym-yme(j), sj,
     $                    -1, em, pr, cosbi, sinbi, -cosbj, sinbj,
     $                    ass, asn, ans, ann, bss, bsn, bns, bnn)
            end if
            if (ksym .eq. 3)                   then
              call   COEFF (xi, yi, 2.*xsym-xme(j), 2.*ysym-yme(j), sj,
     $                    1, em, pr, cosbi, sinbi, -cosbj, -sinbj,
     $                    ass, asn, ans, ann, bss, bsn, bns, bnn)
            end if
            call    SETUP  (i, j, kod(j), g, ss0, sn0,
     $                    ass, asn, ans, ann, bss, bsn, bns, bnn,
     $                    b, c, r, 2*numbe, MAXEQS)
 2500   continue
 3000 continue
        print *, 'Influence coefficient matrix & RHS vector generated'
        if ( op .eq. 'STORE' ) then
C    --- STORE data
C    --- install dataset
          call  GMPUNT ( ldi, 'COEFF ', idsn, 16, 100 )
C    --- construct record name & write data
          n = 2*numbe
          igap = MAXEQS - n
          call   GMCORN ( rname, 'C ', 1, n )
          call   GMPUTN ( 'W', ldi, idsn, rname, 'S', c, -n,
     $                    0, igap, 0, 200 )
C    --- install dataset
          call   GMPUNT ( ldi, 'RHS ', idsn, 16, 500 )
C    --- construct record name & write data
          n = 2*numbe
          call   GMCORN ( rname, 'R ', 1, n )
          call   GMPUTN ( 'W', ldi, idsn, rname, 'S', r, n,
     $                    0, 0, 0, 600 )
          call   GMFLUB ( ldi, 0, 2000 )
          return
        end if

C --- LOAD data
 4000 continue
C --- find dataset
        idsn = LMFIND ( ldi, 'COEFF ', 1000 )
        if ( idsn .eq. 0 ) then
          print*, ' Cannot find COEFF dataset; nothing LOADed.'
        else
C    --- get record name cycles, construct record name & read data
          n = MAXEQS**2
          call   GMGECY ( ' ', ldi, idsn, 'C ', nrec, ilow,
     $                    ihigh, 1100 )
          igap = MAXEQS - nrec
          call   GMCORN ( rname, 'C ', ilow, ihigh )
          call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', c,
     $                    n, 0, igap, 0, 1200 )
        end if
C --- find dataset
        idsn = LMFIND ( ldi, 'RHS ', 1500 )
        if ( idsn .eq. 0 ) then
          print*, ' Cannot find RHS dataset; nothing LOADed.'
        else
C    --- get record name cycles, construct record name & read data
          n = MAXEQS
          call   GMGECY ( ' ', ldi, idsn, 'R ', nrec, ilow,
     $                    ihigh, 1600 )
          call   GMCORN ( rname, 'R ', ilow, ihigh )
          call   GMGETN ( 'R/L', ldi, idsn, rname, 'S', r,
     $                    n, 0, 0, 0, 1700 )
        end if
*
        return
        end
```

```
*
*       Solve for unknown boundary values
```

```
*
      subroutine  SOLVE ( op )
*
      implicit      none
      include       'database.inc'
      include       'element.inc'
      character*(*) op
      character*20  rname
      integer       ising
      integer       n, nrec, ilow, ihigh, LMFIND, idsn
*
      if ( op .eq. 'LOAD' ) go to 1000
      call  GAUSSER (c, r, x, 2*numbe, MAXEQS, ising)
      if (ising .eq. 0)     then
        print *, 'Discrete equations solved'
        if ( op .eq. 'STORE' ) then
C     --- STORE data
C     --- install dataset
          call  GMPUNT ( ldi, 'SOLUTION ', idsn, 16, 500 )
C     --- construct record name & write data
          n = 2*numbe
          call  GMCORN ( rname, 'X ', 1, n )
          call  GMPUTN ( 'W', ldi, idsn, rname, 'S', x, n,
     $                   0, 0, 0, 600 )
          call  GMFLUB ( ldi, 0, 2000 )
        end if
      else
        print *, 'Singularity detected at BE equation',ising
      end if
      return
*
C --- LOAD data
 1000 continue
C --- find dataset
      idsn = LMFIND ( ldi, 'SOLUTION ', 1500 )
      if ( idsn .eq. 0 ) then
        print*, ' Cannot find SOLUTION dataset; nothing LOADed.'
      else
C   --- get record name cycles, construct record name & read data
        n = MAXEQS
        call  GMGECY ( ' ', ldi, idsn, 'X ', nrec, ilow,
     $                 ihigh, 1600 )
        call  GMCORN ( rname, 'X ', ilow, ihigh )
        call  GMGETN ( 'R/L', ldi, idsn, rname, 'S', x,
     $                 n, 0, 0, 0, 1700 )
      end if
*
      return
      end
```

Subroutine GENERATE calls COEFF (which is essentially the same as a TWOBI subroutine with the same name) and SETUP, which fills the entries of the influence coefficient matrix and right-hand-side vector:

```
*
*     Calculate source/receiver coefficients
*
      subroutine  COEFF
     $       (xi, yi, xj, yj, aj,
     $        msym, em, pr, cosbi, sinbi, cosb, sinb,
     $        ass, asn, ans, ann, bss, bsn, bns, bnn)
*
      implicit       none
      real           xi, yi, xj, yj, aj
      real           em, pr, cosbi, sinbi, cosb, sinb
      real           ass, asn, ans, ann, bss, bsn, bns, bnn
      real           pi, con, pr1, pr2, pr3
      integer        msym
      real           cma, cpa, cxb, cyb, cosg, sing
      real           r1s, r2s, fl1, fl2
      real           tb1, tb2, tb3, tb4, tb5
      real           asst, asnt, anst, annt
      real           bsst, bsnt, bnst, bnnt
*
      pi =   4.*atan2(1.,1.)
      con =  1.0/(4.*pi*(1.-pr))
      pr1 =  1.-2*pr
      pr2 =  2.*(1.-pr)
      pr3 =  3.-4.*pr
      cxb =  (xi-xj)*cosb + (yi-yj)*sinb
      cyb = -(xi-xj)*sinb + (yi-yj)*cosb
      cosg = cosbi*cosb + sinbi*sinb
      sing = sinbi*cosb - cosbi*sinb
*
      cma =  cxb - aj
      cpa =  cxb + aj
      r1s =  cma**2 + cyb**2
      r2s =  cpa**2 + cyb**2
      fl1 =   0.5*log(r1s)
      fl2 =   0.5*log(r2s)
      tb2 =  -con*(fl1-fl2)
      tb3 =  con*(atan2(cpa,cyb)-atan2(cma,cyb))
      tb1 =  -cyb*tb3 + con*(cma*fl1-cpa*fl2)
      tb4 =   con*(cyb/r1s-cyb/r2s)
      tb5 =   con*(cma/r1s-cpa/r2s)
*
      asst =   pr2*cosg*tb3 + pr1*sing*tb2 + cyb*(sing*tb4+cosg*tb5)
      asnt =  -pr1*cosg*tb2 + pr2*sing*tb3 + cyb*(cosg*tb4-sing*tb5)
      anst =  -pr2*sing*tb3 + pr1*cosg*tb2 + cyb*(cosg*tb4-sing*tb5)
      annt =   pr1*sing*tb2 + pr2*cosg*tb3 - cyb*(sing*tb4+cosg*tb5)
*
      bsst =   pr3*cosg*tb1 + cyb*(sing*tb2-cosg*tb3)
      bsnt =   pr3*sing*tb1 + cyb*(cosg*tb2+sing*tb3)
      bnst =  -pr3*sing*tb1 + cyb*(cosg*tb2+sing*tb3)
      bnnt =   pr3*cosg*tb1 - cyb*(sing*tb2-cosg*tb3)
*
      ass =    ass + msym*asst
      asn =    asn + asnt
      ans =    ans + msym*anst
      ann =    ann + annt
*
      bss =    bss + msym*bsst
      bsn =    bsn + bsnt
      bns =    bns + msym*bnst
      bnn =    bnn + bnnt
      return
      end
```

```
*
*     Set up influence coeff matrix and RHS of discrete system
*
      subroutine  SETUP
     $            (i, j, bckodj, g, ss0, sn0,
     $             ass, asn, ans, ann,
     $             bss, bsn, bns, bnn,
     $             b, c, r, n, nc)
*
      implicit      none
      integer       i, j, n, nc, bckodj
      real          ss0, sn0, g, bs, bn
      real          ass, asn, ans, ann, bss, bsn, bns, bnn
      real          b(*), c(nc,*), r(*)
*
      if (bckodj .eq. 0)          then
        c(2*i-1,2*j-1) =  ass
        c(2*i-1,2*j  ) =  asn
        c(2*i  ,2*j-1) =  ans
        c(2*i  ,2*j  ) =  ann
        bs =   0.5*(b(2*j-1)-ss0)/g
        bn =   0.5*(b(2*j  )-sn0)/g
        r(2*i-1) =  r(2*i-1) + bss*bs + bsn*bn
        r(2*i  ) =  r(2*i  ) + bns*bs + bnn*bn
      else if (bckodj .eq. 1)     then
        c(2*i-1,2*j-1) =  -bss
        c(2*i-1,2*j  ) =  -bsn
        c(2*i  ,2*j-1) =  -bns
        c(2*i  ,2*j  ) =  -bnn
        r(2*i-1) =  r(2*i-1) - ass*b(2*j-1) - asn*b(2*j)
        r(2*i  ) =  r(2*i  ) - ans*b(2*j-1) - ann*b(2*j)
      else if (bckodj .eq. 2)     then
        c(2*i-1,2*j-1) =  -bss
        c(2*i-1,2*j  ) =   asn
        c(2*i  ,2*j-1) =  -bns
        c(2*i  ,2*j  ) =  ann
        bn =   0.5*(b(2*j  )-sn0)/g
        r(2*i-1) =  r(2*i-1) - ass*b(2*j-1) + bsn*bn
        r(2*i  ) =  r(2*i  ) - ans*b(2*j-1) + bnn*bn
      else
        c(2*i-1,2*j-1) =  ass
        c(2*i-1,2*j  ) =  -bsn
        c(2*i  ,2*j-1) =  ans
        c(2*i  ,2*j  ) =  -bnn
        bs =   0.5*(b(2*j-1)-ss0)/g
        r(2*i-1) =  r(2*i-1) + bss*bs - asn*b(2*j)
        r(2*i  ) =  r(2*i  ) + bns*bs - ann*b(2*j)
      end if
      return
      end
```

Subroutine SOLVE calls GAUSSER, which is a naïve implementation of unsymmetric Gauss elimination without pivoting:

```
*
*     Solve algebraic equation system A x = b by Gauss elimination
*
      subroutine  GAUSSER
     $            (a, b, x, n, na, ising)
*
      implicit      none
      integer       n, na, ising
      real          a(na,*), b(*), x(*), c, sum
      integer       i, j, k
*
      ising = 0
      do 2000  j = 1,n-1
        if (a(j,j) .eq. 0.0)   then
          ising =  j
          return
        end if
        do 1500  k = j+1,n
          c =   a(k,j)/a(j,j)
          do 1400  i = j,n
            a(k,i) = a(k,i) - c*a(j,i)
 1400       continue
          b(k) = b(k) - c*b(j)
 1500      continue
 2000    continue
*
       x(n) =  b(n)/a(n,n)
       do 3000  j = n-1,1,-1
         sum = 0.0
         do 2500  i = j+1,n
           sum =  sum + a(j,i)*x(i)
 2500      continue
         x(j) = (b(j)-sum)/a(j,j)
 3000    continue
       return
       end
```

(The only redeeming quality about **GAUSSER** is that the code is quite short; in fact, it's about the shortest possible implementation of a linear equation solver.)

THIS PAGE LEFT BLANK INTENTIONALLY.

9. Printing Data

One area in which interactive operation excels is data display. If you are using an interactive Processor for a engineering design task, you can selectively trim the otherwise voluminous output to the important essentials. Conversely, if you are debugging a new or modified implementation, you may want more output than is normally required; for example printing the influence coefficient matrix.

What goes for printed output applies with equal force to graphic output. We are not going to illustrate graphic displays here, however, since the details depend strongly on the output device and the plotting software you are using.

The `PRINT` command is similar to the `DEFINE` command in that it takes a second keyword that specifies what is to be printed:

`SEGMENTS`	Prints segment geometry data and number of elements per segment.
`BOUNDARY_CONDITIONS`	Prints boundary condition (BC) code and prescribed boundary values for each segment.
`SYMMETRY_CONDITIONS`	Prints symmetry conditions if any in effect.
`MATERIAL`	Prints material property data.
`PRESTRESS`	Prints prestress data.
`FIELD_LOCATIONS`	Prints information about output-location lines if any is defined.
`ELEMENTS`	Prints detailed boundary-element data produced by subroutine `BUILD` (this is primarily for debugging).
`COEFFICIENTS`	Prints the matrix **C** of influence coefficients assembled by `GENERATE` (this is primarily for debugging).
`RHS`	Prints the right-hand side (forcing) vector **r** assembled by `GENERATE` (this is primarily for debugging).
`SOLUTION`	Prints the solution vector **x** calculated by `SOLVE` (this is primarily for debugging).
`RESULTS`	Print stresses and displacements at boundary-element midpoints or at output field locations, depending on a command qualifier.

In this section none of the subroutines load or store data. However, in a real Processor the results, such as displacements and stresses, which are computed under the `PRINT RESULTS` command would be stored. They would be stored because these results are often plotted or reordered for tabulation. The plots and tables are used to study the results or for inclusion in a report or as presentation. By now you should be able to modify `PRINT RESULTS`, §9.3, to `LOAD` and `STORE` data for the post-processing activities described above. Be brave give it a try.

The `PRINT` command is processed by subroutine `PRINT`, which has a "case" structure similar to that of subroutine `DEFINE`:

```
*
*     Interpret PRINT command
*
      subroutine    PRINT
*
      implicit      none
      character     key*8, CCLVAL*8
      integer       ICLTYP
      logical       CMATCH
*
      if (ICLTYP(2) .le. 0)  then
        call   CLREAD (' PRINT what? ',
     $     ' BOUNDARY, ELEMENTS, COEFFICIENTS,'//
     $      'FIELD, MATERIAL, PRESTRESS&&'//
     $      'RESULTS, RHS, SOLUTION, SYMMETRY')
        key =  CCLVAL(1)
      else
        key =  CCLVAL(2)
      end if
      if (CMATCH (key, 'B^OUNDARY'))    then
        call  PRINT_BOUNDARY_CONDITIONS
      else if (CMATCH (key, 'C^OEFFICIENTS') .or.
     $          CMATCH (key, 'I^NFLUENCE')) then
        call  PRINT_INFLUENCE_COEFFICIENTS
      else if (CMATCH (key, 'E^LEMENTS')) then
        call  PRINT_ELEMENTS
      else if (CMATCH (key, 'F^IELD'))   then
        call  PRINT_FIELD_LOCATIONS
      else if (CMATCH (key, 'M^ATERIAL')) then
        call  PRINT_MATERIAL
      else if (CMATCH (key, 'P^RESTRESS')) then
        call  PRINT_PRESTRESS
      else if (CMATCH (key, 'RE^SULTS'))  then
        call  PRINT_RESULTS
      else if (CMATCH (key, 'RHS'))       then
        call  PRINT_RHS_VECTOR
      else if (CMATCH (key, 'SE^GMENT'))  then
        call  PRINT_SEGMENTS
      else if (CMATCH (key, 'S^OLUTION')) then
        call  PRINT_SOLUTION_VECTOR
      else if (CMATCH (key, 'SY^MMETRY')) then
        call  PRINT_SYMMETRY_CONDITIONS
      else
        print *,'*** Illegal or ambiguous keyword '.key.' after PRINT'
      end if
      return
      end
```

Subroutine PRINT provides our second example of an implementation that *prompts for missing data.* See DB OPEN, §6.0, for another example. If you type only the keyword PRINT followed by a carriage return, you will see the prompt

```
Print what?
```

on the screen, and you are supposed to type the next keyword, *e.g.*, SEGMENTS that you forgot. (Notice that this friendly technique was not used for the DEFINE command explained in §7.0; instead subroutine DEFINE complains about missing keywords after DEFINE.)

Next we examine the subordinate routines.

9.1 Printing Input Data

The implementation of the subroutines that print segment, boundary condition, symmetry, material, prestress, and field-location data is straightforward and so are simply listed next as a group:

```
*
*     Print segment data
*
      subroutine   PRINT_SEGMENTS
*
      implicit      none
      include       'segment.inc'
      integer       i, k
*
      k =    0
      do 2000  i = 1,MAXSEG
        if (segdef(i) .gt. 0)      then
          if (k .eq. 0)            then
            print '(/A/A6,A9,4A12)',
     $            ' Boundary Segment Data',
     $            'Segm', 'Elements', 'Xbeg', 'Ybeg', 'Xend', 'Yend'
          end if
          k =    k + 1
          print '(I6,I9,3X,4G12.4)',
     $           i, numel(i), xbeg(i), ybeg(i), xend(i), yend(i)
        end if
 2000   continue
      if (k .eq. 0)          then
        print *, 'Segment tables are empty'
      end if
      print *, ' '
      return
      end
```

```
*
*     Print boundary data in response to a PRINT BOUNDARY command
*
      subroutine   PRINT_BOUNDARY_CONDITIONS
*
      implicit      none
      include       'segment.inc'
*
      integer       i, k
      character*9   given(0:3)
      data    given /'SS and NS', 'SD and ND', 'SD and NS', 'SS and ND'/
*
      k =    0
      do 2000  i = 1,MAXSEG
        if (segdef(i) .gt. 0)     then
```

```
          if (k .eq. 0)           then
            print '(/A/A6,A11,2A12)',
     $           ' Boundary Conditions Data', 'Segm',
     $           'Given',  'Shear', 'Normal'
          end if
          k =    k + 1
          print '(I5,1X,A11,3X,1P2G12.3)',
     $          i, given(kode(i)), bvs(i), bvn(i)
        end if
 2000   continue
*
      if (k .eq. 0)           then
        print *, 'Boundary tables are empty'
      end if
      print *, ' '
      return
      end
*
*     Print symmetry data
*
      subroutine   PRINT_SYMMETRY_CONDITIONS
*
      implicit      none
      include       'symmetry.inc'
*
      print '(/A)', ' Symmetry Data'
      if (ksym .eq. 3)            then
        print *, 'Symmetry about axis X=',xsym
        print *, '             and axis Y=',ysym
      else if (ksym .eq. 1)       then
        print *, 'Symmetry about axis X=',xsym
      else if (ksym .eq. 2)       then
        print *, 'Symmetry about axis Y=',ysym
      else
        print *, 'No symmetry conditions'
      end if
      print *, ' '
      return
      end
*
*     Print material property data
*
      subroutine   PRINT_MATERIAL
*
      implicit      none
      include       'material.inc'
*
      print '(/A)', ' Material Property Data'
      print '('' Elastic modulus:'',1PE12.3)', em
      print '('' Poisson''''s ratio:'',F12.3)', pr
      print *, ' '
      return
      end
*
*     Print field location data
*
      subroutine   PRINT_FIELD_LOCATIONS
*
      implicit      none
      include      'output.inc'
      integer       i, k
*
      k =     0
      do 2000  i = 1,MAXLIN
        if (lindef(i) .gt. 0)      then
          if (k .eq. 0)            then
            print '(/A/A6,A9,A9,3A12)',
     $            ' Field Location Data',
     $            'Line', 'Int.Pts', 'x-first', 'y-first',
     $            'x-last', 'y-last'
          end if
          k =    k + 1
          print '(I6,I9,4G12.4)',
     $           i, nintop(i), xfirst(i), yfirst(i), xlast(i), ylast(i)
        end if
 2000   continue
*
      if (k .eq. 0)          then
        print *, 'FIELD Location Tables are empty'
      end if
      print *, ' '
      return
      end
```

9.2 Debug-Oriented Print Commands

The PRINT ELEMENTS, PRINT COEFFICIENTS, PRINT RHS and PRINT SOLUTION are detailed print commands primarily useful in debug situations. They are implemented in the following subroutines:

```
*
*     Print detailed boundary element data
*
      subroutine  PRINT_ELEMENTS
*
      implicit      none
*     include       'segment.inc'
      include       'element.inc'
      integer        m
*
      if (numbe .le. 0)           then
        print *, 'Boundary element table empty'
        return
      end if
      print '(/A/A5,A8,2A11,A12,A8,A9,A12)',
     $      ' Boundary Element Data',
     $      'Elem', 'Xmid', 'Ymid', 'Length',
     $      'Orient', 'BCode', 'Shear', 'Normal'
      do 2000  m = 1,numbe
        print '(I5,1P3G11.3,0PF10.2,I6,1P2G12.3)',
     $          m,xme(m),yme(m),2.*hleng(m),
     $          (180./3.14159265)*atan2(sinbet(m),cosbet(m)),
     $          kod(m), b(2*m-1),b(2*m)
 2000   continue
      print *, ' '
      return
      end
```

```
*
*     Print influence coefficient matrix
*
      subroutine  PRINT_INFLUENCE_COEFFICIENTS
*
      implicit      none
*     include       'segment.inc'
      include       'element.inc'
*
      print '(/A)', ' Influence Coefficient Matrix'
      call   PRINT_REAL_MATRIX (c, MAXEQS, 2*numbe, 2*numbe)
      print *, ' '
      return
      end
*
```

```
*      Print right hand side vector
*
      subroutine PRINT_RHS_VECTOR
      implicit      none
*     include       'segment.inc'
      include       'element.inc'
*
      print '(/A)', ' Right Hand Side (Forcing) Vector'
      call   PRINT_REAL_MATRIX (r, 1, 1, 2*numbe)
      print *, ' '
      return
      end
```

```
*
*      Print right hand side vector
*
      subroutine PRINT_SOLUTION_VECTOR
      implicit      none
*     include       'segment.inc'
      include       'element.inc'
*
      print '(/A)', ' Solution Vector'
      call   PRINT_REAL_MATRIX (x, 1, 1, 2*numbe)
      print *, ' '
      return
      end
```

The last three subroutines call PRIMATRIX, which is a "no frills" array printer:

```
*
*      Print real matrix (or vector) in 6-column template
*
      subroutine PRINT_REAL_MATRIX
     $      (a, na, m, n)
      integer   na, m, n, i, j, jref
      real      a(na,*)
      do 4000  jref = 0,n-1,6
        print '(1X,6I12)', (j,j=jref+1,min(jref+6,n))
        do 3000  i = 1,m
          print '(I4,1P6E12.4)', i,(a(i,j),j=jref+1,min(jref+6,n))
 3000     continue
 4000   continue
       return
       end
```

9.3 Printing Results

The PRINT RESULTS command without a qualifier lists stresses and displacements computed at boundary element midpoints. If qualifier FIELD appears, the command refers to the field points previously defined. This switch is implemented in subroutine PRINT_RESULTS:

```
*
*      Process PRINT RESULTS command
*
       subroutine   PRINT_RESULTS
*
       implicit      none
       integer       ICLSEQ
*
       if (ICLSEQ(3,'F^IELD') .eq. 0)    then
         call  PRINT_BOUNDARY_RESULTS
       else
         call  PRINT_FIELD_RESULTS
       end if
       return
       end
```

The code above provides an example of the use of ICLSEQ [1], §5.3, to test for the existence of a specific qualifier, in this case FIELD.

9.3.1 Printing Boundary Results

This is done by subroutine PRINT_BOUNDARY_RESULTS, the implementation of which is straightforward:

```
*
*      Print stresses and displacement @ boundary element midpoints
*
       subroutine  PRINT_BOUNDARY_RESULTS
*
       implicit      none
       include       'segment.inc'
       include       'element.inc'
       include       'material.inc'
       include       'prestress.inc'
       integer        k
       real           g, ss0, sn0, sinbi, cosbi
       real           us, un, ux, uy, sign, sigs
*
       print '(/A)', ' Displacements and Stresses at'//
      $              ' Boundary Element Midpoints'
       print '(A5,A9,5A11)', 'Elem', 'u_s', 'u_n', 'u_x', 'u_y',
     $                        'sig_s', 'sig_n'
      g =  0.5*em/(1.+pr)
*
      do 2000  k = 1,numbe
        us =    x(2*k-1)
        un =    x(2*k  )
        sigs =  b(2*k-1)
        sign =  b(2*k  )
        if (kod(k) .eq. 1)       then
          un =  b(2*k-1)
          us =  b(2*k  )
          sigs = x(2*k-1)
          sign = x(2*k)
        else if (kod(k) .eq. 2)  then
          us =  b(2*k-1)
          sigs =  x(2*k-1)
        else if (kod(k) .eq. 3)  then
          un =  b(2*k  )
          sign = b(2*k  )
        end if
        sinbi = sinbet(k)
        cosbi = cosbet(k)
        ux = us*cosbi - un*sinbi
        uy = us*sinbi + un*cosbi
        print '(I5,1P6G11.3)', k, us,un,ux,uy,sigs,sign
 2000   continue
      print *, ' '
      return
      end
```

9.3.2 Printing Field Results

Showing displacement and stresses at field points is complicated by the fact that, unlike finite element programs, such values are not readily available but must be calculated as part of the display procedure. This will become evident as one shows the coding of subroutine PRINT FIELD_RESULTS :

```
*
*     Print stresses and displacements @ specified field points
*
      subroutine  PRINT_FIELD_RESULTS
*
      implicit       none
      include       'output.inc'
      integer        m, p, points
      real           xp, yp, ux, uy, sigxx, sigyy, sigxy, f
      logical        skip
```

```
*
      print '(/A)', ' Displacements and Stresses at'//
     $              ' Specified Field Points'
*
      do 3000  m = 1,MAXLIN
        if (lindef(m) .eq. 0)    go to 3000
        print '(A5,2A10,A8,4A11)', ' Lin', 'x', 'y', 'u_x', 'u_y',
     $                     'sig_xx', 'sig_yy', 'sig_xy'
        points = nintop(m) + 2
        if (xfirst(m) .eq. xlast(m) .and.
     $      yfirst(m) .eq. ylast(m))  points = 1
        f =  0.0
        do 2000  p = 1,points
          if (points .gt. 1)  f =  real(p-1)/(points-1)
          xp = xfirst(m)*(1.0-f) + xlast(m)*f
          yp = yfirst(m)*(1.0-f) + ylast(m)*f
          call  FIELDP (xp, yp, ux, uy, sigxx, sigyy, sigxy, skip)
          if (skip)    then
            print '(I5,2F10.3,6X,A)', m, xp,yp,
     $            'Point is too close to boundary'
          else
            print '(I5,2F10.3,1P5G11.3)', m, xp,yp, ux,uy,
     $                                  sigxx,sigyy,sigxy
          end if
 2000     continue
        print *, ' '
 3000   continue
      return
      end
```

Subroutine FIELDP receives the location XP,YP of the field point and returns the displacement components u_x and u_y, and the stress components σ_{xx}, σ_{yy} and σ_{xy}:

```
*
*     Compute stresses and displacements at field point
*
      subroutine  FIELDP
     $           (xp, yp,
     $            ux, uy, sigxx, sigyy, sigxy, skip)
*
      implicit      none
*     include       'segment.inc'
      include       'element.inc'
      include       'material.inc'
      include       'symmetry.inc'
      include       'prestress.inc'
      real           xp, yp, us, un, ux, uy, sigxx, sigyy, sigxy
      logical        skip
      real           uxus, uxun, uxss, uxsn
      real           uyus, uyun, uyss, uysn
      real           sxxus, sxxun, sxxss, sxxsn
      real           syyus, syyun, syyss, syysn
      real           sxyus, sxyun, sxyss, sxysn
      real           xj, yj, sj, cosbj, sinbj
      real           usj, unj, ssj, snj, ssg, sng
      real           g, ss0, sn0
      integer        j

      skip =  .false.
      ux =  0.0
      uy =  0.0
      sigxx =  sxx0
      sigyy =  syy0
      sigxy =  sxy0
      g =     0.5*em/(1.+pr)

      do 2000  j = 1,numbe
        uxus =  0.0
        uxun =  0.0
        uxss =  0.0
        uxsn =  0.0
        uyun =  0.0
        uyun =  0.0
        uyss =  0.0
        uysn =  0.0
        sxxus = 0.0
        sxxun = 0.0
        sxxss = 0.0
        sxxsn = 0.0
        syyus = 0.0
        syyun = 0.0
        syyss = 0.0
        syysn = 0.0
        sxyus = 0.0
        sxyun = 0.0
        sxyss = 0.0
        sxysn = 0.0
        xj =  xme(j)
        yj =  yme(j)
        sj =  hleng(j)
        if ((xp-xj)**2+(yp-yj)**2 .le. 1.01*(   sj)**2)  then
          skip =  .true.
          return
        end if
        cosbj = cosbet(j)
        sinbj = sinbet(j)
        ss0 = (syy0-sxx0)*sinbj*cosbj + sxy0*(cosbj**2-sinbj**2)
        sn0 = sxx0*sinbj**2 - 2.*sxy0*sinbj*cosbj + syy0*cosbj**2
        call   SOMIGLIANA (xp, yp, xj, yj, sj,
     $              1, em, pr, cosbj, sinbj,
     $              uxus, uxun, uxss, uxsn,
     $              uyus, uyun, uyss, uysn,
     $              sxxus, sxxun, sxxss, sxxsn,
     $              syyus, syyun, syyss, syysn,
     $              sxyus, sxyun, sxyss, sxysn)
        if (ksym .eq. 1 .or. ksym .eq. 3)  then
          call   SOMIGLIANA (xp, yp, 2.*xsym-xme(j), yj, sj,
     $                -1, em, pr, cosbj, -sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        end if
        if (ksym .eq. 2 .or. ksym .eq. 3)  then
          call   SOMIGLIANA (xp, yp, xj, 2.*ysym-yme(j), sj,
     $                -1, em, pr, -cosbj, sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        end if
        if (ksym .eq. 3)                    then
          call   SOMIGLIANA (xp, yp, 2.*xsym-xme(j), 2.*ysym-yme(j), sj,
     $                1, em, pr, -cosbj, -sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        end if
        usj = x(2*j-1)
        unj = x(2*j  )
        ssj = b(2*j-1) - ss0
        snj = b(2*j  ) - sn0
        if (kod(j) .eq. 1)   then
          usj =  b(2*j-1)
          unj =  b(2*j  )
          ssj =  x(2*j-1)
          snj =  x(2*j  )
        else if (kod(j) .eq. 2)  then
          usj =  b(2*j-1)
          ssj =  x(2*j  )
        else if (kod(j) .eq. 3)  then
```

```
          unj =  b(2*j  )
          snj =  x(2*j  )
        end if
        ssg = 0.5*ssj/g
        sng = 0.5*snj/g
        ux =  ux + uxus*usj + uxun*unj + uxss*ssg + uxsn*sng
        uy =  uy + uyus*usj + uyun*unj + uyss*ssg + uysn*sng
        usj = 2.*g*usj
        unj = 2.*g*unj
        sigxx = sigxx + sxxus*usj + sxxun*unj + sxxss*ssj + sxxsn*snj
        sigyy = sigyy + syyus*usj + syyun*unj + syyss*ssj + syysn*snj
        sigxy = sigxy + sxyus*usj + sxyun*unj + sxyss*ssj + sxysn*snj
 2000   continue
      return
      end
```

Finally, FIELDP calls subroutine SOMIGLIANA to evaluate the important boundary-on-field-point influence coefficients:

```
*
*     Calculate field influence coefficients from Somigliana's formula
*
      subroutine  SOMIGLIANA
     $    (x, y, xj, yj, aj, msym, em, pr, cosb, sinb,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
*
      implicit       none
      real            x, y, xj, yj, aj, em, pr, cosb, sinb
      real            uxus, uxun, uxss, uxsn
      real            uyus, uyun, uyss, uysn
      real            sxxus, sxxun, sxxss, sxxsn
      real            syyus, syyun, syyss, syysn
      real            sxyus, sxyun, sxyss, sxysn
      integer         msym
      real            pi, con, pr1, pr2, pr3
      real            cxb, cyb, cosg, sing, cpa, cma
      real            r1s, r2s, fl1, fl2
      real            tb1, tb2, tb3, tb4, tb5, tb6, tb7
      real            uxust, uxunt, uxsst, uxsnt
      real            uyust, uyunt, uysst, uysnt
      real            sxxust, sxxunt, sxxsst, sxxsnt
      real            syyust, syyunt, syysst, syysnt
      real            sxyust, sxyunt, sxysst, sxysnt
      real            cosb2, sinb2, cos2b, sin2b
*
```

```
      pi =   4.*atan2(1.,1.)
      con =  1.0/(4.*pi*(1.-pr))
      pr1 =  1.-2*pr
      pr2 =  2.*(1.-pr)
      pr3 =  3.-4.*pr
*
      cxb =  (x-xj)*cosb + (y-yj)*sinb
      cyb = -(x-xj)*sinb + (y-yj)*cosb
*
      cma =  cxb - aj
      cpa =  cxb + aj
      r1s =  cma**2 + cyb**2
      r2s =  cpa**2 + cyb**2
      fl1 =   0.5*log(r1s)
      fl2 =   0.5*log(r2s)
      tb2 =  -con*(fl1-fl2)
      tb3 =  con*(atan2(cpa,cyb)-atan2(cma,cyb))
      tb1 =  -cyb*tb3 + con*(cma*fl1-cpa*fl2)
      tb4 =   con*(cyb/r1s-cyb/r2s)
      tb5 =   con*(cma/r1s-cpa/r2s)
      tb6 =  con*((cma**2-cyb**2)/r1s**2-(cpa**2-cyb**2)/r2s**2)
      tb7 =  -con*2.*cyb*(cma/r1s**2-cpa/r2s**2)
*
      uxust =  pr1*sinb*tb2 - pr2*cosb*tb3 + cyb*(sinb*tb4-cosb*tb5)
      uxunt =  pr1*cosb*tb2 + pr2*sinb*tb3 - cyb*(cosb*tb4+sinb*tb5)
      uxsst =  pr3*cosb*tb1 - cyb*(sinb*tb2+cosb*tb3)
      uxsnt = -pr3*sinb*tb1 + cyb*(cosb*tb2-sinb*tb3)
      uyust = -pr1*cosb*tb2 - pr2*sinb*tb3 - cyb*(cosb*tb4+sinb*tb5)
      uyunt =  pr1*sinb*tb2 - pr2*cosb*tb3 - cyb*(sinb*tb4-cosb*tb5)
      uysst =  pr3*sinb*tb1 + cyb*(cosb*tb2-sinb*tb3)
      uysnt =  pr3*cosb*tb1 + cyb*(sinb*tb2+cosb*tb3)
*
      cosb2 = cosb*cosb
      sinb2 = sinb*sinb
      cos2b = cosb2-sinb2
      sin2b = 2.*sinb*cosb
*
      sxxust = 2.*cosb2*tb4 + sin2b*tb5 - cyb*(cos2b*tb6-sin2b*tb7)
      syyust = 2.*sinb2*tb4 - sin2b*tb5 + cyb*(cos2b*tb6-sin2b*tb7)
      sxyust = sin2b*tb4 - cos2b*tb5 - cyb*(sin2b*tb6+cos2b*tb7)
      sxxunt = -tb5 - cyb*(sin2b*tb6+cos2b*tb7)
      syyunt = -tb5 + cyb*(sin2b*tb6+cos2b*tb7)
      sxyunt =  cyb*(cos2b*tb6-sin2b*tb7)
      sxxsst = -tb2 - pr2*(cos2b*tb2-sin2b*tb3)
     $         + cyb*(cos2b*tb4+sin2b*tb5)
      syysst = -tb2 - pr2*(cos2b*tb2-sin2b*tb3)
     $         - cyb*(cos2b*tb4+sin2b*tb5)
      sxysst = - pr2*(sin2b*tb2+cos2b*tb3)
```

```
$             + cyb*(sin2b*tb4-cos2b*tb5)
  sxxsnt  = -tb3 + pr1*(sin2b*tb2+cos2b*tb3)
$             + cyb*(sin2b*tb4-cos2b*tb5)
  syysnt  = -tb3 - pr1*(sin2b*tb2+cos2b*tb3)
$             - cyb*(sin2b*tb4-cos2b*tb5)
  sxysnt  = - pr1*(cos2b*tb2-sin2b*tb3)
$             - cyb*(cos2b*tb4+sin2b*tb5)
*
  uxus =    uxus + msym*uxust
  uxun =    uxun + uxunt
  uxss =    uxss + msym*uxsst
  uxsn =    uxsn + uxsnt
  uyus =    uyus + msym*uyust
  uyun =    uyun + uyunt
  uyss =    uyss + msym*uysst
  uysn =    uysn + uysnt
*
  sxxus =    sxxus + msym*sxxust
  sxxun =    sxxun + sxxunt
  sxxss =    sxxss + msym*sxxsst
  sxxsn =    sxxsn + sxxsnt
  syyus =    syyus + msym*syyust
  syyun =    syyun + syyunt
  syyss =    syyss + msym*syysst
  syysn =    syysn + syysnt
  sxyus =    sxyus + msym*sxyust
  sxyun =    sxyun + sxyunt
  sxyss =    sxyss + msym*sxysst
  sxysn =    sxysn + sxysnt
*
  return
  end
```

The DBEM2 Processor is complete.

10. DBEM2 Structure

After all the coding details given in §5.0 through §9.0 it is perhaps refreshing to get an overall picture of the structure of DBEM2. A *hierarchical diagram* of the module structure provides a portion of the picture:

```
DBEM2
  DOCOMMAND
    BUILD
    CLEAR
    CLOSE
    DEFINE
      DEFINE_BOUNDARY_CONDITIONS
        BCVALUES
      DEFINE_ELEMENTS
      DEFINE_MATERIAL
      DEFINE_FIELD_LOCATIONS
      DEFINE_PRESTRESS
      DEFINE_SEGMENTS
      DEFINE_SYMMETRY
    GENERATE
      COEFF
      SETUP
    OPEN
    PRINT
      PRINT_BOUNDARY_CONDITIONS
      PRINT_BOUNDARY_RESULTS
      PRINT_COEFFICIENTS
        PRINT_REAL_MATRIX
      PRINT_ELEMENTS
      PRINT_FIELD_LOCATIONS
      PRINT_FIELD_RESULTS
        FIELDP
          SOMIGLIANA
      PRINT_MATERIAL
      PRINT_PRESTRESS
      PRINT_RHS
        PRINT_REAL_MATRIX
      PRINT_SEGMENTS
      PRINT_SOLUTION
        PRINT_REAL_MATRIX
      PRINT_SYMMETRY
    SOLVE
      GAUSSER
    STOP
```

This diagram of course excludes the NICE utilities such as the CLIP and GAL-DBM system. With this omission noted, the deepest module level is five. This is a feature symptomatic of a fairly simple Processor. (Actual production Processors in the NICE system reach module levels of order 15–20.)

Another part of the picture is provided by a diagram of the DBEM2 GAL Library structure of datasets and associated record names:

```
DBEM2 GAL LIBRARY
  BCVALUES
    BVN
    BVS
    KODE
  COEFF
    C
  ELEMENT
    B
    COSBET
    HLENG
    KOD
    NUMBE
    SINBET
    XME
    YME
  FIELD
    LINDEF
    NINTOP
    XFIRST
    XLAST
    YFIRST
    YLAST
  MATERIAL
    EM
    PR
  PRESTRESS
    SXXO
    SXYO
    SYYO
  RHS
    R
```

```
SEGMENT
   NUMEL
   SEGDEF
   XBEG
   XEND
   YBEG
   YEND
SOLUTION
   X
SYMMETRY
   KSYM
   XSYM
   YSYM
```

Of course not all of the datasets will appear in all Libraries; only the datasets corresponding to the data that are stored appears.

A diagram that shows the interaction between the database and the Processor modules provides a connection between the first two diagrams. In the diagram below the Processor module commands are shown on the left and the corresponding GAL Library datasets are shown on the left. The $\longleftrightarrow$ symbol indicates that the data in the designated dataset is either loaded or stored within the designated module. The $\longleftarrow$ symbol indicates that the corresponding dataset must be loaded before the designated command should be entered. By searching the diagram for the $\longleftrightarrow$ symbol you can see which command along with a `LOAD` sub-command must be entered to load the desired dataset.

```
DEFINE
  DEFINE_SEGMENTS ←→ Dataset:  SEGMENT
  DEFINE_ELEMENTS ←→ Updates Record NUMEL in Dataset:  SEGMENT
  DEFINE_BOUNDARY_CONDITIONS ←→ Dataset:  BCVALUES
  DEFINE_MATERIAL ←→ Dataset:  MATERIAL
  DEFINE_SYMMETRY ←→ Dataset:  SYMMETRY
  DEFINE_PRESTRESS ←→ Dataset:  PRESTRESS
  DEFINE_FIELD_LOCATIONS ←→ Dataset:  FIELD
BUILD ←— Dataset:  SEGMENT
      ←— Dataset:  BCVALUES
      ←→ Dataset:  ELEMENT
GENERATE ←— Dataset:  ELEMENT
         ←— Dataset:  MATERIAL
         ←— Dataset:  SYMMETRY
         ←— Dataset:  PRESTRESS
         ←→ Dataset:  COEFF
         ←→ Dataset:  RHS
SOLVE ←— Dataset:  COEFF
      ←— Dataset:  RHS
      ←— Dataset:  ELEMENT
      ←→ Dataset:  SOLUTION
PRINT
  PRINT_BOUNDARY_CONDITIONS ←— Dataset:  BCVALUES
  PRINT_BOUNDARY_RESULTS ←— Dataset:  MATERIAL
                         ←— Dataset:  ELEMENT
                         ←— Dataset:  SOLUTION
  PRINT_COEFFICIENTS ←— Dataset:  ELEMENT
                     ←— Dataset:  COEFF
  PRINT_ELEMENTS ←— Dataset:  ELEMENT
  PRINT_FIELD_LOCATIONS ←— Dataset:  FIELD
  PRINT_FIELD_RESULTS ←— Dataset:  FIELD
                      ←— Dataset:  PRESTRESS
                      ←— Dataset:  MATERIAL
                      ←— Dataset:  ELEMENT
                      ←— Dataset:  SYMMETRY
                      ←— Dataset:  SOLUTION
  PRINT_MATERIAL ←— Dataset:  MATERIAL
  PRINT_PRESTRESS ←— Dataset:  PRESTRESS
  PRINT_RHS ←— Dataset:  ELEMENT
            ←— Dataset:  RHS
  PRINT_SEGMENTS ←— Dataset:  SEGMENT
  PRINT_SOLUTION ←— Dataset:  ELEMENT
                 ←— Dataset:  SOLUTION
  PRINT_SYMMETRY ←— Dataset:  SYMMETRY
```

Now we have enough information on how the Processor and the GAL-DBM work together to discuss some potential extensions to this simple Processor.

- To make the Processor more "user gentle" you can add some checks to determine if the data required at this stage have been entered or loaded. If not you can then tell the user what is missing and what needs to be done to rectify the situation. To implement this you will need a data structure with flags to indicate whether the data have been entered or loaded. And you will need a corresponding table that contains the commands needed to enter or load the data.

  You can get very elaborate if you let the user indicate whether the data are to be loaded or entered at the keyboard. Then based on the user response you can search the open GAL Library for the proper dataset and load it, or jump to the proper subroutine for entering the data from the keyboard.

  You can even enter the realm of artificial intelligence (AI). You would develop some rules for what order the data are entered, what data are needed for what command, what commands perform what tasks, etc. Then with a simple forward chaining inference engine you can assist the user at any point in the analysis by telling them what usual comes next, what data are needed, how to get to some point in the analysis from where the user asks, etc.

- To enter a research mode or maximum flexibility mode you may wish to break the DBEM2 Processor into several independently executable Processors. A good starting point would be pre-processing -- all of the `DEFINE` commands, `BUILD`, `GENERATE`, `SOLVE`, and post-processing -- all of the `PRINT` commands. Now you could have other boundary elements that would be incorporated in the `BUILD` process. Or somewhat easier, you can replace the `SOLVE` Processor with a new solver, `SOLVE_NEW`, and compare the performance of the two solvers. Of course output data generated by the post-processing Processor can be stored on the database. Then you could develop plot Processors to display the data.

  You can be very ambitious and combine the boundary element method with a finite element code. Here you would need to develop some special matrix Processors to properly assemble the system matrix. Or be very brave and try a coupled solution procedure.

Anyway, I hope you can see the unlimited potential of developing computational software in this mode. A common command language and a common database manager to unify the software is a very powerful paradigm.

"From little acorns the mighty oak does grow."

THIS PAGE LEFT BLANK INTENTIONALLY.

11. An Example Problem

It is convenient to test DBEM2 on the same example problem used by Crouch and Starfield [4]. The problem concerns a unit-radius circular hole in an infinite body under uniaxial tension at infinity. The boundary element discretization for one-quarter of the hole is shown in Figure 11-1.

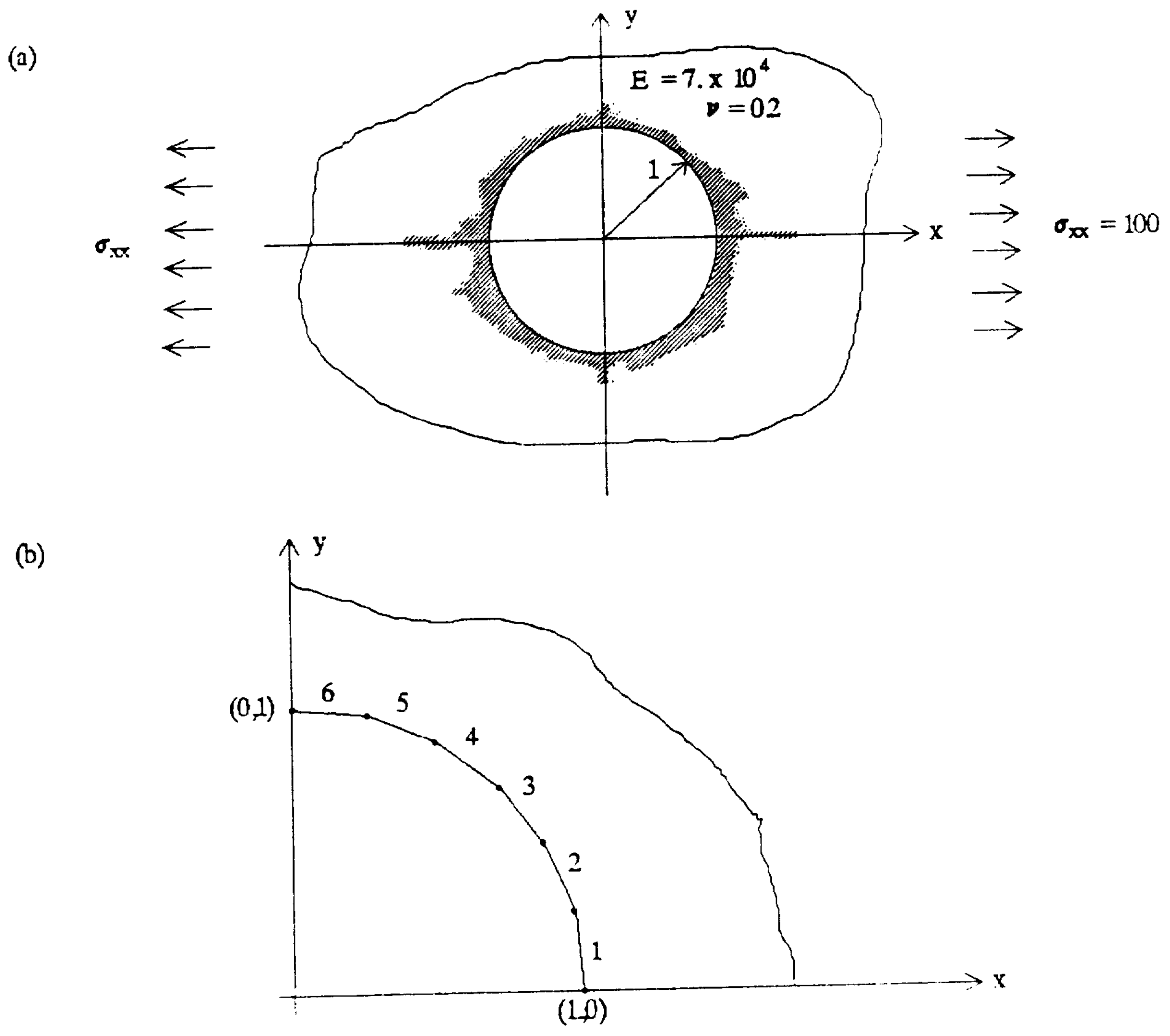

Figure 11-1. Circular hole in an infinite body:
(a) problem specifications, (b) boundary element model

Both $x = 0$ and $y = 0$ are symmetry lines. The boundary contour is approximated by six straight-line segments, each of which consists of one boundary element. Two field point lines are chosen along portions of the x and y axes as shown in Figure 11-1(a).

The input for this problem is prepared (with the text editor) in the form of a script command file:

```
open lib=circhole.gal
clear
def segments
  seg=1 b=1,0         e=.9659,.2588
  seg=2 b=.9659,.2588 e=.8660,.5000
  seg=3 b=.8660,.5000 e=.7071,.7071
  seg=4 b=.7071,.7071 e=.5000,.8660
  seg=5 b=.5000,.8660 e=.2588,.9659
  seg=6 b=.2588,.9659 e=0,1
  store
  end
def material
  em=7.E4 ; pr=0.2 ; store ; end
def symmetry
  xsym=0 ; ysym = 0 ; store ; end
def prestress
  sxx0=100 ; store ; end
def field
  line=1 f=1,0 l=6,0 p=9
  line=2 f=0,1 l=0,6 p=9
  store
  end
pri seg ; pri mat ; pri bou ; pri symm ; pri pres ; pri field
build/store ; gen/store ; sol/store
pri res ; pri res/field
```

Note that there is no need for DEFINE ELEMENT input data because each segment contains only one boundary element, which is the default assumption.

Upon starting the DBEM2 processor, this file is inserted in the command stream through an ADD directive [3], §13.1. For example, under UNIX:

```
> dbem2
  DBEM2> *add circhole.add
```

where circhole.add is the assumed name of the input file. The printed results should then be compared with those given on Appendix C of Crouch and Starfield [4].

Here is what you would see on your screen.

```
 Tables initialized


Boundary Segment Data
 Segm Elements        Xbeg         Ybeg         Xend         Yend
    1        1       1.000          0.        0.9659       0.2588
    2        1      0.9659       0.2588       0.8660       0.5000
    3        1      0.8660       0.5000       0.7071       0.7071
    4        1      0.7071       0.7071       0.5000       0.8660
```

```
     5        1      0.5000      0.8660      0.2588      0.9659
     6        1      0.2588      0.9659      0.          1.000


Material Property Data
Elastic modulus:   7.000E+04
Poisson's ratio:       0.200


Boundary Conditions Data
 Segm      Given        Shear        Normal
   1    SS and NS        0.           0.
   2    SS and NS        0.           0.
   3    SS and NS        0.           0.
   4    SS and NS        0.           0.
   5    SS and NS        0.           0.
   6    SS and NS        0.           0.


Symmetry Data
 Symmetry about axis X=  0.
            and axis Y=  0.


Prestress (Initial Field Stresses) Data
Sigma_xx:     100.        Sigma_yy:       0.        Sigma_xy:       0.

Field Location Data
 Line  Int.Pts  x-first      y-first       x-last       y-last
   1        9   1.000          0.          6.000          0.
   2        9     0.         1.000           0.         6.000

Discrete model building completed:     6 boundary elements

 Influence coefficient matrix & RHS vector generated
 Discrete equations solved

Displacements and Stresses at Boundary Element Midpoints
Elem      u_s         u_n         u_x         u_y       sig_s       sig_n
   1 -4.768E-04 -2.649E-03  2.689E-03 -1.267E-04     0.          0.
   2 -1.302E-03 -2.177E-03  2.509E-03 -3.697E-04     0.          0.
   3 -1.778E-03 -1.359E-03  2.161E-03 -5.836E-04     0.          0.
   4 -1.778E-03 -4.142E-04  1.663E-03 -7.539E-04     0.          0.
   5 -1.302E-03  4.038E-04  1.048E-03 -8.712E-04     0.          0.
   6 -4.767E-04  8.761E-04  3.582E-04 -9.309E-04     0.          0.

Displacements and Stresses at Specified Field Points
 Lin        x          y      u_x          u_y      sig_xx   sig_yy   sig_xy
```

```
  1    1.000     0.000       Point is too close to boundary
  1    1.500     0.000   2.116E-03  5.483E-11  64.8     -42.0     -3.078E-06
  1    2.000     0.000   1.668E-03  1.026E-10  72.3     -18.1     -2.897E-06
  1    2.500     0.000   1.365E-03  1.255E-10  80.2     -10.2     -3.334E-07
  1    3.000     0.000   1.151E-03  2.173E-10  85.5     -6.54     -1.722E-09
  1    3.500     0.000   9.935E-04  2.065E-10  89.0     -4.57      2.339E-06
  1    4.000     0.000   8.733E-04  1.669E-10  91.4     -3.38      1.921E-06
  1    4.500     0.000   7.787E-04  3.464E-11  93.1     -2.61     -2.656E-07
  1    5.000     0.000   7.024E-04  1.132E-10  94.4     -2.08      1.784E-07
  1    5.500     0.000   6.396E-04  4.157E-10  95.3     -1.70     -1.803E-07
  1    6.000     0.000   5.870E-04  7.854E-11  96.0     -1.41      2.896E-06

 Lin       x         y      u_x        u_y      sig_xx   sig_yy    sig_xy
  2    0.000     1.000       Point is too close to boundary
  2    0.000     1.500  -2.307E-11 -2.615E-03  167.      28.7     -9.665E-07
  2    0.000     2.000  -5.174E-11 -5.075E-03  130.      21.4     -7.445E-07
  2    0.000     2.500   4.023E-11 -6.942E-03  117.      15.1      6.003E-07
  2    0.000     3.000  -6.035E-11 -8.478E-03  111.      11.0      1.752E-07
  2    0.000     3.500   2.717E-10 -9.789E-03  108.      8.33      1.715E-06
  2    0.000     4.000  -1.864E-10 -1.094E-02  106.      6.49     -1.311E-06
  2    0.000     4.500  -1.627E-10 -1.195E-02  105.      5.19     -1.560E-06
  2    0.000     5.000  -7.970E-11 -1.287E-02  104.      4.24      5.272E-07
  2    0.000     5.500  -2.068E-12 -1.371E-02  103.      3.53      4.855E-07
  2    0.000     6.000   5.215E-10 -1.447E-02  102.      2.98      6.761E-07
```

Now lets take a look at the database that was generated during this run. All of the commands that begin with * are directives. The directives are described in [3].

```
DBEM2> *toc

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+  Library  1     File: circhole.gal                                        +
+  Form: GAL82    File size:      1973 words     No. of Datasets:   9       +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Seq#     Date       Time     Lk Records  Processor Dataset name
  1   06:01:88  20:07:20    0       36  DBEM2     SEGMENT
  2   06:01:88  20:07:20    0        2  DBEM2     MATERIAL
  3   06:01:88  20:07:20    0        3  DBEM2     SYMMETRY
  4   06:01:88  20:07:20    0        3  DBEM2     PRESTRESS
  5   06:01:88  20:07:20    0       12  DBEM2     FIELD
  6   06:01:88  20:07:20    0       49  DBEM2     ELEMENT
  7   06:01:88  20:07:20    0       12  DBEM2     COEFF
  8   06:01:88  20:07:20    0       12  DBEM2     RHS
  9   06:01:88  20:07:20    0       12  DBEM2     SOLUTION
```

Here we see the table of contents (toc) for the GAL Library that was generated during the run. You can see that the file name is circhole.gal, the size of the file, there are 9 datasets, and other information, like the date and time the data were stored.

To see the second level of the database, the records, the *print rat or *rat directive [3], §49.4, can be used. Lets take a look at some of the record structure. You may wish to compare this output to the second diagram in §10.0 or compare with the discussion in §3.

```
DBEM2> *rat 1,1

Record Table of dataset SEGMENT
Key          L_cyc H_cyc  Type Log_size
NUMEL            1     6   I          1
SEGDEF           1     6   I          1
XBEG             1     6   S          1
XEND             1     6   S          1
YBEG             1     6   S          1
YEND             1     6   S          1

DBEM2> *rat 1,2

Record Table of dataset MATERIAL
Key          L_cyc H_cyc  Type Log_size
EM               0     0   S          1
PR               0     0   S          1

DBEM2> *rat 1,7

Record Table of dataset COEFF
Key          L_cyc H_cyc  Type Log_size
C                1    12   S         12
```

The first number after the *rat is the GAL Library ldi, which is printed as the Library in the *toc [3], §49.1, output shown above. The second number is the dataset sequence number, also shown in the table of contents output. The *rat output tells us the names of the records, the Key; the number of records, from L_cyc to H_cyc; the data type, I is integer — S is floating point; and the logical size of each record. For example, the record key NUMEL in the dataset SEGMENT has six records containing one integer in each record. The record keys in the dataset MATERIAL are a bit different in that each record only has one floating point number, so there are no cycles. The record key C in the dataset COEFF has 12 records each containing 12 floating point numbers. This is the square system coefficient matrix.

We can also look at the data stored within each record key.

```
DBEM2> *print rec 1,1,NUMEL.1:6
 Record NUMEL.1 of dataset SEGMENT
 1:          1
 Record NUMEL.2 of dataset SEGMENT
 1:          1
 Record NUMEL.3 of dataset SEGMENT
 1:          1
 Record NUMEL.4 of dataset SEGMENT
```

```
   1:             1
   Record NUMEL.5 of dataset SEGMENT
   1:             1
   Record NUMEL.6 of dataset SEGMENT
   1:             1
 DBEM2> *print rec 1,2,EM
  Record EM of dataset MATERIAL
  1:   7.0000E+04
 DBEM2> *print rec 1,2,PR
  Record PR of dataset MATERIAL
  1:   2.0000E-01
 DBEM2> *print rec 1,7,C.7
  Record C.7 of dataset COEFF
  1: 4.6007E-04 -2.4613E-02  2.0957E-03 -3.1559E-02  1.2205E-02 -6.3592E-02
  7: 4.5021E-01  4.1663E-03  1.1955E-02  7.2579E-02  1.4381E-03  4.3028E-02
```

Here we have used the *print record directive [3], §49.2 & §49.3, to show: 1) the values in the six records in the record key NUMEL, 2) the values of the elastic modulus, EM, and Poisson's ratio, PR, in the dataset MATERIAL, and 3) the seventh column of the system coefficient matrix, C, which is stored in the dataset COEFF.

Now to exit the DBEM2 Processor we type **stop** to produce the following on our screen:

```
 DBEM2> stop
 <DM> CLOSE, Ldi:  1, File: circhole.gal
   Hope you enjoyed the ride!
 <CL> PNS exhausted
      ENDRUN called by CLIP
```

A few days later we decide we would like to solve the same problem, but we wish to increase the number of elements to 2 for each segment to see how much this improves the results. So here is what it looks like.

```
aml_9: 17 > dbem2
 DBEM2> clear
  Tables initialized
 DBEM2> open lib=circhole.gal
 <DM> OPEN,  Ldi:  1, File: circhole.gal , Attr: old, Block I/O
 DBEM2> *toc

 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +  Library  1    File: circhole.gal                                          +
 +  Form: GAL82   File size:     1973 words     No. of Datasets:   9          +
 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 Seq#    Date      Time     Lk Records  Processor Dataset name
    1  06:01:88 20:07:20    0      36  DBEM2     SEGMENT
    2  06:01:88 20:07:20    0       2  DBEM2     MATERIAL
    3  06:01:88 20:07:20    0       3  DBEM2     SYMMETRY
  4  06:01:88 20:07:20   0      3  DBEM2     PRESTRESS
  5  06:01:88 20:07:20   0     12  DBEM2     FIELD
  6  06:01:88 20:07:20   0     49  DBEM2     ELEMENT
  7  06:01:88 20:07:20   0     12  DBEM2     COEFF
  8  06:01:88 20:07:20   0     12  DBEM2     RHS
  9  06:01:88 20:07:20   0     12  DBEM2     SOLUTION

DBEM2> def seg ; load ; end
DBEM2> def mat ; load ; end
DBEM2> def sym ; load ; end
DBEM2> def pres ; load ; end
DBEM2> def field ; load ; end
DBEM2> pri seg

Boundary Segment Data
 Segm Elements        Xbeg        Ybeg        Xend        Yend
    1        1       1.000          0.      0.9659      0.2588
    2        1      0.9659      0.2588      0.8660      0.5000
    3        1      0.8660      0.5000      0.7071      0.7071
    4        1      0.7071      0.7071      0.5000      0.8660
    5        1      0.5000      0.8660      0.2588      0.9659
    6        1      0.2588      0.9659          0.       1.000

DBEM2> pri pres

Prestress (Initial Field Stresses) Data
Sigma_xx:     100.       Sigma_yy:       0.        Sigma_xy:       0.

DBEM2> pri mat

Material Property Data
Elastic modulus:   7.000E+04
Poisson's ratio:       0.200
 DBEM2> help def elem
 <DBEM2>
    DEFINE
       ELEMENTS

       The DEFINE ELEMENTS command introduces subordinate commands
       that specify into how many boundary elements segments are to
       be subdivided.  These commands have the form:

          SEG = iseg1, ... isegk   ELEM = ne1, ... nek

       This specifies that segment  iseg1  is to be subdivided into
       ne1 (ge 1) boundary elements,  segment  iseg2  into ne2
       elements, and so on.  Enter the STORE command to store this
       data in the NUMEL record in the SEGMENT dataset.  Enter the
```

```
      LOAD command to load previously defined data from the NUMEL
      record in the SEGMENT dataset.  Terminate these commands
      with an END command.

DBEM2> def elem
Element data> seg = 1:6 elem = 2,2,2,2,2,2
Element data> end
DBEM2> pri seg

Boundary Segment Data
 Segm Elements       Xbeg        Ybeg        Xend        Yend
    1        2      1.000          0.      0.9659      0.2588
    2        2     0.9659      0.2588      0.8660      0.5000
    3        2     0.8660      0.5000      0.7071      0.7071
    4        2     0.7071      0.7071      0.5000      0.8660
    5        2     0.5000      0.8660      0.2588      0.9659
    6        2     0.2588      0.9659          0.       1.000

DBEM2> build
Discrete model building completed:   12 boundary elements

DBEM2> generate
 Influence coefficient matrix & RHS vector generated
DBEM2> solve
 Discrete equations solved
DBEM2> pri res

Displacements and Stresses at Boundary Element Midpoints
Elem       u_s        u_n        u_x        u_y       sig_s      sig_n
   1 -4.791E-04 -2.755E-03  2.794E-03 -1.152E-04       0.         0.
   2 -1.288E-03 -2.289E-03  2.438E-03 -9.782E-04       0.         0.
   3 -1.769E-03 -1.483E-03  2.047E-03 -1.067E-03       0.         0.
   4 -1.743E-03 -5.670E-04  1.191E-03 -1.394E-03       0.         0.
   5 -1.265E-03  1.977E-04  6.131E-04 -1.124E-03       0.         0.
   6 -3.329E-04  4.904E-04 -1.865E-04 -5.626E-04       0.         0.
   7 -1.523E-03 -5.829E-04  1.563E-03 -4.646E-04       0.         0.
   8 -1.513E-03 -2.601E-04  1.359E-03 -7.147E-04       0.         0.
   9 -1.306E-03  3.710E-04  1.064E-03 -8.424E-04       0.         0.
  10 -1.107E-03  5.584E-04  8.093E-04 -9.396E-04       0.         0.
  11 -5.808E-04  9.193E-04  4.557E-04 -9.873E-04       0.         0.
  12 -3.178E-04  9.855E-04  1.864E-04 -1.019E-03       0.         0.

DBEM2> pri res /field

Displacements and Stresses at Specified Field Points
 Lin      x        y      u_x        u_y      sig_xx   sig_yy    sig_xy
   1    1.000    0.000      Point is too close to boundary
   1    1.500    0.000  1.935E-03  1.465E-11    55.5    -53.9   -1.108E-06
  1   2.000    0.000  1.432E-03 -5.107E-11   70.7     -17.6    -4.039E-06
  1   2.500    0.000  1.142E-03  1.263E-10   80.0     -8.45    -3.628E-06
  1   3.000    0.000  9.520E-04  2.523E-10   85.6     -4.90     1.794E-06
  1   3.500    0.000  8.165E-04  5.708E-10   89.1     -3.20     1.658E-06
  1   4.000    0.000  7.150E-04  6.382E-10   91.5     -2.25     2.942E-07
  1   4.500    0.000  6.360E-04  4.635E-10   93.2     -1.68    -1.418E-06
  1   5.000    0.000  5.727E-04  5.684E-10   94.5     -1.30    -4.365E-07
  1   5.500    0.000  5.208E-04  7.305E-10   95.4     -1.04     2.026E-06
  1   6.000    0.000  4.776E-04  5.825E-10   96.1    -0.853     2.337E-06

Lin      x        y      u_x         u_y     sig_xx   sig_yy    sig_xy
  2   0.000    1.000      Point is too close to boundary
  2   0.000    1.500 -3.197E-11 -5.243E-03   160.      24.8     4.456E-07
  2   0.000    2.000  7.015E-11 -9.904E-03   126.      18.4    -7.870E-08
  2   0.000    2.500 -1.200E-10 -1.347E-02   114.      13.0    -1.185E-06
  2   0.000    3.000  1.169E-10 -1.640E-02   109.      9.48     1.955E-06
  2   0.000    3.500  9.394E-11 -1.889E-02   107.      7.17    -7.394E-07
  2   0.000    4.000 -3.967E-10 -2.107E-02   105.      5.60    -2.295E-06
  2   0.000    4.500  3.040E-10 -2.301E-02   104.      4.48     3.884E-07
  2   0.000    5.000 -2.895E-10 -2.474E-02   103.      3.66    -1.996E-06
  2   0.000    5.500 -7.609E-11 -2.632E-02   103.      3.04    -6.769E-07
  2   0.000    6.000  5.487E-10 -2.777E-02   102.      2.57     1.350E-06
DBEM2> stop
<DM> CLOSE, Ldi:  1, File: circhole.gal
  Hope you enjoyed the ride!
<CL> PHS exhausted
     ENDRUN called by CLIP
```

This interactive session starts off with the usual **clear** and **open** commands. Then to make sure that I have the correct GAL Library, the *toc [3], §49.1, is used. Everything looks okay, so the problem definition commands to **define segments, materials, symmetry, prestress**, and **field locations** are issued with the subcommand **load** to load these data from the GAL Library. The **print segments** command is used to look at the old segment data that were just loaded — gives you a warm feeling to see that the data are really there. Just for insurance I check the prestress data and the material data. Now, I want to enter data to use 2 elements per segment, but I can't remember the proper syntax, so I use the **help define elements** command to get the on-line help (see [3], Appendix H for a discussion of help files). Now, I enter the data and check it with another **print segments** command. Note that, I now have 2 elements for each segment. Then the three number crunching modules, **build, generate**, and **solve**, are brought into action to obtain the new solution. Finally, the new results are printed. Note that, I did not store any of the new problem data. Only a simple change was made to compare answers. If big changes were made, I would have stored the new data.

Experiment with your own changes to this problem. Then try some new problems. Enjoy!

THIS PAGE LEFT BLANK INTENTIONALLY.

12. References

1 Felippa, Carlos A., *The Computational Structural Mechanics Testbed Architecture: Volume III The Interface*, NASA CR-178386 October 1988.

2 Wright, Mary A., Regelbrugge, Marc E., and Felippa, Carlos A., *The Computational Structural Mechanics Testbed Architecture: Volume IV - The Global-Database Manager GAL-DBM.*, NASA CR-178387 October 1988.

3 Felippa, Carlos A. and Underwood, Philip, *The Computational Structural Mechanics Testbed Architecture: Volume II - The Directives*, NASA CR-178385 October 1988.

4 Crouch, S. L. and Starfield, A. M., *Boundary Element Methods in Solid Mechanics: with Applications in Rock Mechanics and Geological Engineering*. G. Allen and Unwin, London, 1983.

THIS PAGE LEFT BLANK INTENTIONALLY.

NASA

Report Documentation Page

1. Report No. NASA CR-181732	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Application Developer's Tutorial for the CSM Testbed Architecture		5. Report Date October 1988
		6. Performing Organization Code
7. Author(s) Phillip Underwood and Carlos A. Felippa		8. Performing Organization Report No. LMSC-D878511
9. Performing Organization Name and Address Lockheed Missiles and Space Company, Inc. Research and Development Division 3251 Hanover Street Palo Alto, California 94304		10. Work Unit No. 505-63-01-10
		11. Contract or Grant No. NAS1-18444
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Phillip Underwood, Lockheed Missiles and Space Company, Inc.
Palo Alto, California 94304

Carlos A. Felippa, Center for Space Structures and Controls,
University of Colorado, Boulder, CO 80309-0429

Langley Technical Monitor: W. Jefferson Stroud

16. Abstract

This tutorial serves as an illustration of the use of the programmer interface of the CSM Testbed Architecture (NICE). It presents a complete, but simple, introduction to using both the GAL-DBM (Global Access Library-Database Manager) and CLIP (Command Language Interface Program) to write a NICE processor. Familiarity with the CSM Testbed architecture is required.

17. Key Words (Suggested by Authors(s)) Structural analysis software; Command language interface software; Data management software	18. Distribution Statement Unclassified- -Unlimited; Subject Category 39

19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 101	22. Price A06

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171